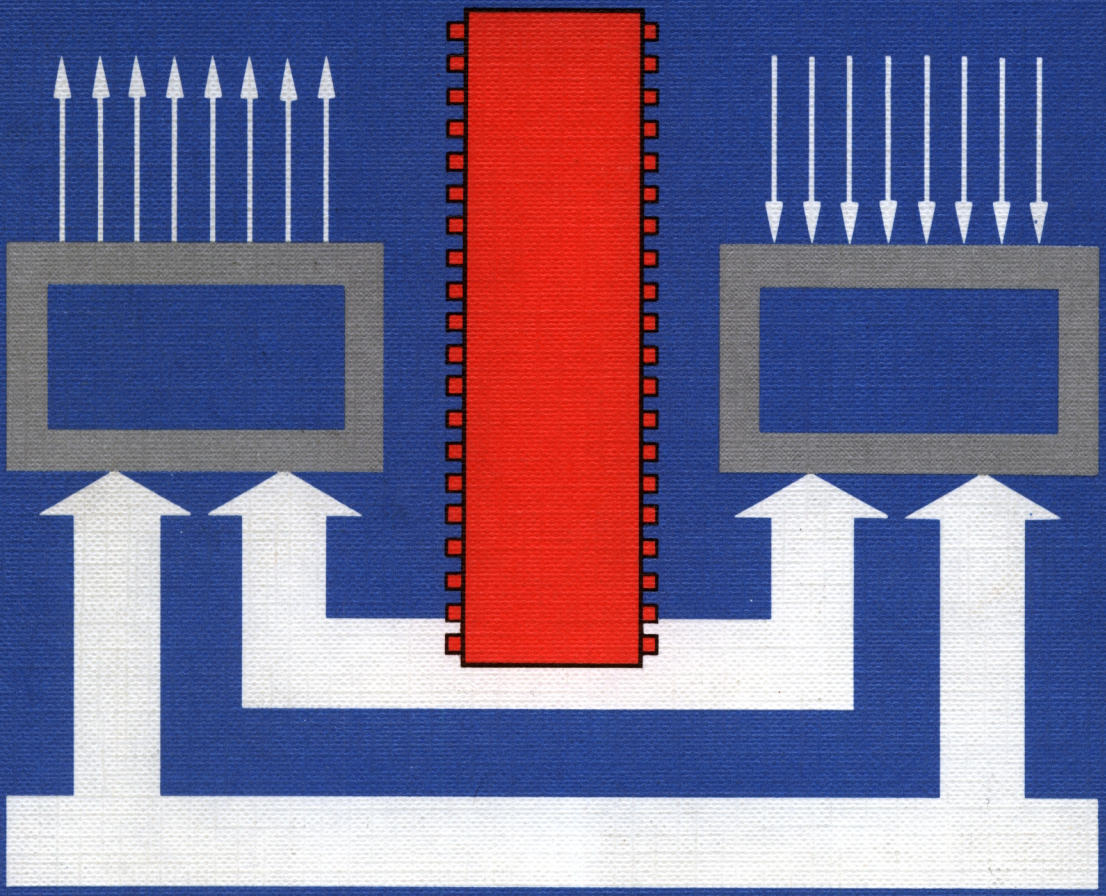
Franzis Elektronik
ohne Ballast

Immerzeel

Mikrocomputer ohne Ballast



Franzis'

Immerzeel

Mikrocomputer ohne Ballast

In der Gruppe
Franzis Elektronik ohne Ballast
sind erschienen:

Fellbaum/Loos, Phonotechnik ohne Ballast
Fellbaum/Loos, HiFi-Technik ohne Ballast
Limann, Elektronik ohne Ballast
Limann, Fernsehtechnik ohne Ballast
Limann, Funktechnik ohne Ballast
Warnke, Tonbandtechnik ohne Ballast
Kloss, Leistungselektronik ohne Ballast

Franzis Elektronik ohne Ballast

Martinus Bernardus Immerzeel

Mikrocomputer ohne Ballast

Ein Mikrocomputer-Anleitungsbuch für Anfänger mit Assemblerprogrammen für die CPU 6502

Mit 125 Abbildungen und 46 Tabellen

Franzis'

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Immerzeel, Martinus Bernardus
Mikrocomputer ohne Ballast: e. Mikrocomputer-Anleitungsbuch für Anfänger mit Assemblerprogrammen für d. CPU 6502 / Martinus Bernardus Immerzeel. [Übertr. aus d. Niederländ.: Otto Gothe]. – München: Franzis, 1982.
(Franzis-Elektronik ohne Ballast)
Einheitssacht.: Micro computers van A tot Z <dt.>
ISBN 3-7723-6981-2

© 1981 De Muiderkring B.V. – Bussum. Titel der Originalausgabe: Micro Computers van A tot Z
Deutsche Ausgabe: ©1982 Franzis-Verlag GmbH, München.
Übertragung aus dem Niederländischen: Otto Gothe.
Technische Bearbeitung: Dipl.-Ing. (FH) Herwig Feichtinger.
Jeder Nachdruck, auch auszugsweise, und jede Wiedergabe der Abbildungen, auch in verändertem Zustand, sind verboten.
Satz: SatzStudio Pfeifer, Germering
Druck: Franzis-Druck GmbH, Karlstraße 35, 8000 München 2

ISBN 3-7723-6981-2

Vorwort

Das Arbeiten mit einem Mikrocomputer erfordert eingehende Kenntnisse digitaler Rechenweise, exakter Logik sowie binärer und dezimaler Codierung. Obwohl über diese Themen bereits ausführlich geschrieben wurde, halte ich es doch für angebracht, auf diese Probleme hinzuweisen im Hinblick auf die Leser, die sich noch nicht mit diesen Dingen beschäftigten. Im übrigen beschränke ich diese Abhandlung auf ein unbedingt notwendiges Maß sowie auf ihre Bedeutung für Mikrocomputer, so daß sich ein Studium der betreffenden Kapitel als notwendig erweist. Außerdem dient die Entwicklung eines Problems bis zum Programmschema als Vorbereitung für den Hauptabschnitt „Der Mikrocomputer".

Dieser Hauptabschnitt ist zielbewußt auf den Mikroprozessor 6502 abgestimmt. Dieser ist in zahlreichen Mikrocomputern eingesetzt (z.B. KIM-1, AIM-65, CBM, SYM-1, Challenger, PC-100) und enthält nicht weniger als 13 Adressierungsarten, was ihn als besonders interessant erscheinen läßt. Um jedoch nicht völlig einseitig zu werden und Unterschiede zwischen den Prozessoren angeben zu können, werden noch die Mikroprozessoren 6800 und 8080A beschrieben. Hierzu gehört auch noch eine Aufstellung über die Befehlssätze der genannten Prozessoren.

Die meisten der mir bekannten Bücher gehen davon aus, den Leser durch Konfrontation mit einem vollständigen Programm mit dem Programmieren vertraut zu machen. Ich halte das nicht für angebracht, sondern der Leser sollte vielmehr einen Einblick in das Problem erhalten, das der Computer mit dem Programm lösen soll.

Gewöhnlich sind für die verschiedenen Fachgebiete eine Anzahl Basiselemente erforderlich, die, in entsprechender Weise kombiniert, ein geschlossenes Ganzes bilden, vergleichbar mit einer Anzahl elektronischer Bauelemente, die man für den Aufbau eines Rundfunkgerätes benötigt. In Übereinstimmung hiermit habe ich eine Reihe von Basisprogrammen ausgewählt, die als Programmteile zu einem vollständigen Programm zusammengestellt werden können.

Außer der Bearbeitung der Basisprogramme enthält das Buch noch ein vollständiges Programm. Das aufgeführte Problem kann bei jedem Leser als bekannt vorausgesetzt werden (Quizprogramm). Damit soll die Kombination einer Anzahl von Basisprogrammen demonstriert werden.

Es ist anzunehmen, daß die Hobbycomputer mit der Programmiersprache Basic eine außerordentliche Verbreitung erfahren werden. Meiner Meinung nach ist ein Buch unvollständig, wenn es nicht wenigstens eine Beschreibung dieser Programmiersprache enthält. Weiterhin ist eine Aufstellung der Fachausdrücke und Abkürzungen erforderlich, womit dieses Buch abgeschlossen wird.

M.B. Immerzeel

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Schaltungen und Verfahren werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden*).

Alle Schaltungen und technischen Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag sieht sich deshalb gezwungen, darauf hinzuweisen, daß er weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen kann. Für die Mitteilung eventueller Fehler sind Autor und Verlag jederzeit dankbar.

*) Bei gewerblicher Nutzung ist vorher die Genehmigung des möglichen Lizenzinhabers einzuholen.

Inhalt

1 Einleitung 9

2 Der analoge und der digitale Computer 11

3 Digitales Rechnen 14
3.1 Zahlensysteme 14
3.2 Umrechnen zwischen Zahlensystemen 15
3.3 Der BCD-Code 17
3.4 Der Hexadezimal-Code 18

4 Rechnen im binären und hexadezimalen System 20
4.1 Binäre Addition 20
4.2 Binäre Subtraktion 21
4.3 Binäres Multiplizieren und Dividieren 21
4.4 Hexadezimales Rechnen 23

5 Rechnen im BCD-System 24

6 Negative Zahlen 27
6.1 Das Zweierkomplement 27
6.2 Rechnen im Zweierkomplement-System 28

7 Logisches Arbeiten 31
7.1 Logische Funktionen UND, ODER, EXCEPT, NICHT 31
7.2 Anwendungen 33

8 Das Programm 35
8.1 Das Flußdiagramm 35
8.2 Sprungbefehle 36
8.3 Programme für logische Funktionen 37
8.4 Der erste Programmansatz 41
8.5 Unterprogramme 44

9 Der Mikrocomputer 46
9.1 Die wichtigsten Bauelemente 46
9.2 Der Speicher 48
9.3 Die CPU 6502 53
9.4 Ein/Ausgabe 60
9.5 Interface 62
9.6 Hintergrundspeicher 65

10 Befehle 70
10.1 Der Befehlssatz 70
10.2 Übertragungsbefehle 70
10.3 Rechenbefehle 79
10.4 Vergleichsbefehle 85
10.5 Logische Befehle 87
10.6 Schiebebefehle 89
10.7 Bedingte Sprungbefehle (Branch) 91
10.8 Unbedingte Sprungbefehle (Jump) 96
10.9 Übrige Befehle 105

11 Nicht indizierte Adressierungsarten . . . 106
11.1 Unmittelbare Adressierung . . . 106
11.2 Zero-Page-Adressierung . . . 107
11.3 Direkte Adressierung . . . 108
11.4 Indirekte Adressierung . . . 109
11.5 Relative Adressierung . . . 110
11.6 Akku und implizite Adressierung . . . 111

12 Indizierte Adressierungsarten . . . 113
12.1 Zero-Page-indizierte Adressierung . . . 113
12.2 Direkt indizierte Adressierung . . . 115
12.3 Indiziert-indirekte Adressierung mit x . . . 116
12.4 Indirekte-indizierte Adressierung . . . 117

13 Einfache Programme . . . 119
13.1 Allgemeines . . . 119
13.2 Einfache Berechnung . . . 123
13.3 Logische Funktionen . . . 124
13.4 Verzögerungszeiten . . . 128
13.5 Lesen der Eingangstore (Ports) . . . 131
13.6 Warteschleifen . . . 135
13.7 Parallel-Serienumsetzung . . . 139
13.8 Pufferspeicher . . . 142
13.9 Das Siebensegment-Display . . . 145
13.10 Multiplikation . . . 148
13.11 Division . . . 152

14 Ein Programmbeispiel . . . 160
14.1 Allgemeines . . . 160
14.2 Wer zuerst drückt . . . 160

15 Die CPU 6800 . . . 174
15.1 Allgemeines . . . 174
15.2 Adressierungsarten . . . 176
15.3 Befehlssatz . . . 177

16 Die CPU 8080A . . . 188
16.1 Allgemeines . . . 188
16.2 Adressierungsarten . . . 195
16.3 Befehlssatz . . . 200

17 Programmiersprachen . . . 202
17.1 Allgemeines . . . 202
17.2 Die Programmiersprache BASIC . . . 204

18 Wortregister . . . 219

19 Abkürzungen . . . 221

Literaturangaben . . . 223

Sachverzeichnis . . . 224

1 Einleitung

In einer Abhandlung über das Programmieren von Computern ist folgender Satz zu lesen:

„Ein Computer ist in erster Linie eine Rechenmaschine."

Beschränken wir uns auf große und kostspielige Maschinen, die in bestimmten Dienststellen und Betrieben eingesetzt sind, dann trifft dieser Satz wohl zu. Einen Hinweis für den Einsatz der Maschine gibt auch schon die Bezeichnung, abgeleitet vom Verbum „to compute", was „rechnen" heißt. Im Prinzip ist der Computer zur Ausführung komplizierter Berechnungen bei hohem Genauigkeitsgrad in kürzester Zeit entwickelt worden.

Aber nicht nur komplizierte Berechnungen lassen sich mit einem Computer durchführen, es können ihm auch Probleme angeboten werden, die der Mensch nicht ohne weiteres lösen kann, da hierfür besondere Rechenmethoden erforderlich sind. Ein Beispiel hierfür ist die sog. „Artillerieaufgabe". Hierbei ist ein Punkt auf der Flugbahn eines sich bewegenden Körpers (z.B. Flugzeug) zu berechnen, an dem sich der Flugkörper und ein abgeschossenes Projektil (Geschoß) treffen sollen.

In *Abb. 1* ist M der Punkt, an dem sich der Flugkörper in dem Augenblick befindet, wo in O das Geschoß abgefeuert wird. Der Auftreffpunkt ist T. Der Abstand MT wird durch Multiplikation der Geschwindigkeit des Flugkörpers mit der Laufzeit des Geschosses über den Abstand OT gefunden. Dieser Abstand ist jedoch unbekannt, da der Punkt T noch berechnet werden muß.

Die Auflösung dieser Aufgabe ist nicht so ohne weiteres zu finden, da durch ständige Lageänderung des Flugkörpers die Berechnungen in so kurzer Zeit durchgeführt werden müssen, woraus sich ergibt, daß infolge des eingeschränkten menschlichen Vermögens zu kurz geschossen wird und nur ein Computer ein richtiges Ergebnis bringen würde.

Ganz allgemein kann ein Computer noch wesentlich mehr Berechnungen ausführen. Neben verwaltungstechnischen Aufgaben kann der Computer für alle Vorgänge, die durch Automaten abgewickelt oder durch Regelsysteme kontrolliert werden, eingesetzt werden.

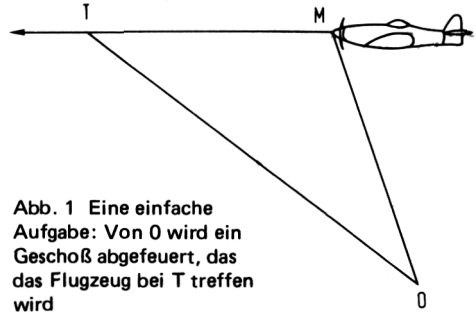


Abb. 1 Eine einfache Aufgabe: Von O wird ein Geschoß abgefeuert, das das Flugzeug bei T treffen wird

Es bedarf keiner besonderen Erläuterungen, da die Industrie nach einer so weit wie möglich erreichbaren Automatisierung bestrebt ist, um zu erkennen, in welchem großen Umfang ein Computer eingesetzt werden kann.

Einsatzmöglichkeiten finden wir z.B. in Operationssälen von Krankenhäusern;
in Walzwerken zur Festlegung des geeigneten Walzenabstandes der Walzstraßen;
zum Regeln chemischer Prozesse;
an Werkzeugmaschinen, wie Drehmaschinen und Bohrwerken;
aber auch im häuslichen Bereich, z.B. an der Nähmaschine, die eine große Anzahl verschiedener Stiche ausführen soll;
bei der Automation von Modelleisenbahnen.

Daß der Computer erst in den letzten Jahren einen Durchbruch erzielen konnte, liegt im wesentlichen an den früher damit verbundenen hohen Kosten. Der Computer enthält eine Vielzahl von Schaltungen mit zahlreichen Bauelementen. Erst integrierte Schaltungen mit großer Packungsdichte, bei denen Tausende von Dioden und Transistoren auf einem Chip zusammengefaßt werden, ermöglichen die Herstellung relativ preisgünstiger Computer. Ebenfalls konnten dadurch Computer kleiner Ausmaße mit bemerkenswerten Möglichkeiten entwickelt werden, nämlich die Mikrocomputer.

Wenn auch der Mikrocomputer nicht die enormen Möglichkeiten seines großen Bruders besitzt, so ist doch die Anzahl der Anwendungsmöglichkeiten erstaunlich.

2 Der analoge und der digitale Computer

Im Schaltbild *Abb. 2* ist ein regelbarer Verstärker angegeben. Hierbei handelt es sich um einen Spannungsverstärker mit hohem Eingangswiderstand in bezug auf R2.

Der Strom durch R2 ergibt sich wie folgt:

$$i_2 = \frac{u_1 - u_i}{R1} + \frac{u_2 - u_i}{R3}$$

Nehmen wir für R1 und R3 Widerstände mit gleichen Werten (R1 = R3), dann finden wir:

$$i_2 = \frac{u_1 + u_2 - 2u_i}{R1}$$

und für die Eingangsspannung u_i

$$u_i = (u_1 + u_2)\,\frac{R2}{R1+2R2}$$

$$u_u = u_1 \cdot A$$

$$u_u = (u_1 + u_2)\,\frac{R2}{R1 + 2R2}\,A$$

Hierin sind sowohl $\frac{R2}{R1 + 2R2}$ als auch A Verhältniszahlen, wobei die Größe von A einstellbar ist. Geben wir A denselben Wert wie $\frac{R1 + 2R2}{R2}$, ergibt sich hieraus:

$$u_u = (u_1 + u_2)\,\frac{R2}{R1 + 2R2} \cdot \frac{R1 + 2R2}{R2}$$

$$\boxed{u_u = u_1 + u_2}$$

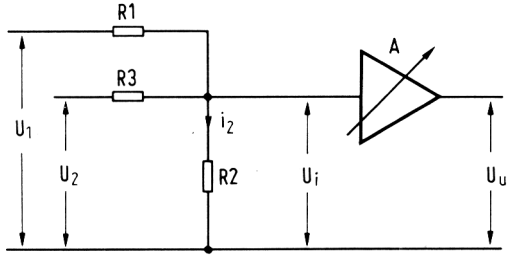


Abb. 2 Addition in einem Analogcomputer

Aus dieser Ableitung folgt, daß mit der angegebenen Schaltung nach Abb. 2 eine Addition durchgeführt werden kann. Dabei sind die Eingangswerte in Spannungswerten anzugeben, aus denen sich das Ausgangsresultat ebenfalls in Form einer Spannung ergibt. Es sind Schaltungen für alle arithmetischen Rechenaufgaben, aber auch für verschiedene Funktionen, so auch trigonometrische Aufgaben möglich. Mit Hilfe dieser Schaltungen, in logischer Anordnung aneinander gereiht, können Berechnungen bestimmter mathematischer Gleichungen ausgeführt werden. Der Nachteil einer derartigen Analog-Rechenmaschine liegt in der ständigen Notwendigkeit einer Abstimmung und Eichung (d.h. Einstellen von $\frac{R1 + 2R2}{R2}$) was in den meisten Fällen nicht durch den Anwender selbst erfolgen kann.

Eine besondere Eigenschaft der Analogtechnik liegt darin, daß eine Größe jeden beliebigen Wert zwischen zwei Grenzwerten annehmen kann. Das ist bei der Digitalrechnung nicht der Fall. Eine veränderliche digitale Größe kann nur schrittweise zu- oder abnehmen. Hierzu als Vergleich eine Analog- bzw. Digitaluhr. Bei einer Analoguhr mit Stunden- und Minutenzeiger sind auch noch Zwischenwerte von Minuten ablesbar. Bei einer Digitaluhr ist das nicht möglich, so daß zur genauen Zeitmessung noch eine Sekundenanzeige erforderlich ist.

Das bedeutet, daß in der Digital-Technik die Genauigkeit dadurch vergrößert werden kann, daß man die Schritte, mit denen die Größen verändert werden, verkleinert. Hier zeigt sich nun deutlich der Unterschied zwischen einem analogen und einem digitalen Computer:

Beim Analog-Computer hängt die Genauigkeit von der Qualität der Schaltung und dem einwandfreien Abgleich ab.
Beim Digital-Computer hängt die Genauigkeit von der Zahl und Größe der Schritte ab.

Daraus ergibt sich, daß sich mit einem Digital-Computer ein über die gesamte Lebensdauer unveränderter höherer Genauigkeitsgrad als mit einem Analog-Computer erreichen läßt.

Es ist noch zu bemerken, daß ein Analog-Computer aus einer Anzahl Recheneinheiten besteht, die abhängig von der Computerfunktion in einer ganz bestimmten Rangfolge angeordnet sein müssen.

Der Digital-Computer dagegen enthält lediglich eine Recheneinheit, die in der Lage sein muß, sämtliche erforderlichen Operationen auszuführen. Diese wird als Zentraleinheit bezeichnet.

Abhängig von der Computerfunktion führt diese Zentraleinheit nacheinander den Arbeitsablauf gemäß einem bestimmten Programm aus, wobei evtl. Zwischenresultate in einem Speicher abgelegt werden können.

Im allgemeinen ist der Analog-Computer nur für eine bestimmte Funktion ausgelegt, da der gegenseitige Zusammenhang der Recheneinheiten nicht oder nur sehr schwer zu ändern ist; das Programm eines Digital-Computers dagegen ist einfach zu verändern, woraus sich eine universale Einsatzmöglichkeit ergibt.

Das ist eine der Ursachen, die dem Digital-Computer einen so großen Vorsprung vor dem Analog-Computer gegeben hat.

Ausgehend von einer Grundversion läßt sich der Digital-Computer für zahlreiche Funktionen umrüsten. Für eine spezifische Funktion ist dann lediglich das erforderliche Programm einzusetzen. Die in den letzten Jahren im Handel erhältlichen Mikrocomputer sind deshalb auch alle Digital-Computer.

3 Digitales Rechnen

3.1 Zahlensysteme

Für das Speichern von Zahlen und für deren Verarbeitung in der Zentraleinheit eines Mikrocomputers sind Register erforderlich. Jedes Register kann eine Zahl einer bestimmten Länge enthalten, d.h. eine bestimmte Anzahl Ziffern.

Die Anwendung des Dezimalsystems führt jedoch zu Schwierigkeiten, da es für den Rechner nicht einfach ist, den verschiedenen Zahlen ebenso viele Werte zuzuschreiben. Es ist daher einfacher, im Dualsystem zu arbeiten. Hier werden nur die Ziffern „0" und „1" gebraucht, mit denen sich außerdem Begriffe leichter ausdrücken lassen, z.B. keine Spannung = „0" und + 5V = „1".

Es folgt nun eine Beschreibung der Zahlensysteme und das Rechnen mit diesen, insbesondere im Hinblick auf die Anwendung bei Mikrocomputern.

Es ist nicht zufällig, daß wir im allgemeinen zum Rechnen das Dezimalsystem benutzen, sind wir doch mit zehn Fingern zum Zählen versehen. Die Bezeichnung „digital" ist ja auch von „Digitus" gleich „Finger" abgeleitet.

Es stehen uns zum Zählen zehn „Symbole" oder Schriftzeichen zur Verfügung:
0, 1, 2, 3, 4, 5, 6 ,7, 8 und 9.

Für größere Zahlen müssen die Schriftzeichen kombiniert werden:
10, 11, 12 . ... 100, 101 usw.
Eigentlich ist es nicht erforderlich, „zehn" für 10, „elf" für 11 usw. auszusprechen, die Zahlen hätten auch mit „eins-null", „eins-eins" usw. benannt werden können.

In einer Zahl hat jede Ziffer auch einen Stellenwert. In der Zahl 583 hat die Ziffer 3 den Stellenwert 3, oder auch $3 \cdot 10^0$, die Ziffer 8 den Wert 80, oder $8 \cdot 10^1$ und die Ziffer 5 den Wert 500 oder $5 \cdot 10^2$. Zwei Beispiele:

$583=500+80+3=5 \cdot 10^2+8 \cdot 10^1+3 \cdot 10^0$

$7542=7000+500+40+2=7 \cdot 10^3+5 \cdot 10^2+4 \cdot 10^1+2 \cdot 10^0$

Im Fünfer-Zahlensystem stehen fünf Symbole zur Verfügung:

0
1
2
3
4 (letztes Symbol)
10 (eins-null und nicht zehn)
11 (eins-eins und nicht elf)
usw.

Im Dualsystem gibt es nur zwei Symbole:

0
1 (letztes Symbol)
10
11
100 (eins-null-null und nicht hundert).

3.2 Umrechnen zwischen Zahlensystemen

Es wurde bereits darauf hingewiesen, daß im Computer nur Zahlen aus dem Dual- oder Binärsystem verwendet werden. Es erscheint daher zweckmäßig, sich mit der Konversion (Umrechnung) von Zahlen aus dem Dezimalsystem in gleichwertige Zahlen des Dualsystems zu befassen.

In der Tabelle sind die ersten sechzehn Zahlen aufgeführt:

Dezimal	*Dual*
0	= 0
$1=2^0$	= 1
$2=2^1$	= 10
$3=2^1+2^0$	= 11
$4=2^2$	= 100
$5=2^2+2^0$	= 101
$6=2^2+2^1$	= 110
$7=2^2+2^1+2^0$	= 111
$8=2^3$	=1000
$9=2^3+2^0$	=1001
$10=2^3+2^1$	=1010
$11=2^3+2^1+2^0$	=1011
$12=2^3+2^2$	=1100
$13=2^3+2^2+2^0$	=1101
$14=2^3+2^2+2^1$	=1110
$15=2^3+2^2+2^1+2^0$	=1111

Völlig übereinstimmend hiermit ist:

Dezimal	*Dual*
$1=2^0=$	1
$2=2^1=$	10
$4=2^2=$	100
$8=2^3=$	1000
$16=2^4=$	10000
$32=2^5=$	100000

usw.

Im Dualsystem folgt daher nach einer 1 eine gleiche Anzahl von Nullen, entsprechend dem Exponenten von 2 im Dezimalsystem.

Die Umrechnung der Zahl 186 verläuft demnach wie folgt:

$186=128+32+16+8+2=2^7+2^5+2^4+2^3+2^1$

Dezimal	*Dual*
$2=2^1=$	10
$8=2^3=$	1000
$16=2^4=$	10000
$32=2^5=$	100000
$128=2^7=$	10000000+
186 =	10111010

$186_{(10)}=10111010_{(2)}$

Der tiefgestellte Index an den Zahlen gibt die Grundzahl des Zahlensystems an.

Umgekehrt:
10111010=10000000+100000+10000+1000+10

Dual	*Dezimal*
$10=2^1=$	2
$1000=2^3=$	8
$10000=2^4=$	16
$100000=2^5=$	32
$10000000=2^7=$	128+
10111010	=186

Im nachfolgenden Beispiel ist eine zweite Umrechnungsart angegeben:

2	186	
2	93+0	a
2	46+1	b
2	23+0	c
2	11+1	d
2	5+1	e
2	2+1	f
2	1+0	g
	0+1	h

die Zahl:

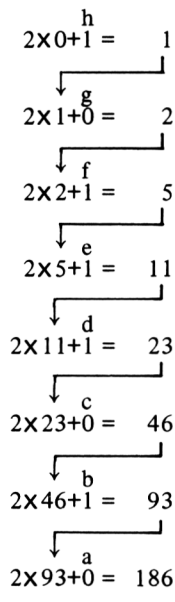
h g f e d c b a
1 0 1 1 1 0 1 0

Die Zahl (hier $186_{(10)}$) wird folglich durch 2 geteilt. Der Rest jeder Teilung ist dann eine 0 oder eine 1, und ist beim niedrigsten Wert angefangen, ein Bit im binären Zahlensystem.

Umgekehrt:

die Zahl:

hgfedcba
10111010



$10111010_{(2)} = 186_{(10)}$

Also immer das Resultat des vorhergehenden Ergebnisses mit 2 multiplizieren, beginnend mit 2X0+1=1.

Im Dualsystem kennt man keine „Ziffern", sondern hier heißt es „Bit" und eine Zahl „Wort". $101_{(2)}$ ist somit ein „Wort", bestehend aus drei Bits, aber $00101_{(2)}$ mit dem gleichen Wert ist ein „Wort", bestehend aus fünf Bits.

3.3 Der BCD-Code

Heute verwendet man bei Mikrocomputern typenabhängig 4-, 8- oder 16-Bit-Register, also immer ein Vielfaches von vier. Mit einem Wort von vier Bit ist es nämlich möglich, jede Ziffer im Dezimalsystem wiederzugeben. ($9_{(10)} = 1001_{(2)}$). Mit der Übertragung der Ziffern in ein „Codewort", das aus den Elementen 1 und 0 besteht, kann man somit jede Zahl aus dem Dezimalsystem in den Computer eingeben. Die Codewörter sind – im folgenden Code als BCD-Code bezeichnet (Binary-Coded Decimal = binär codierte Dezimalzahl) – vom Dezimalsystem abgeleitet.

Dezimal		*BCD*
0	=	0000
1	=	0001
2	=	0010
3	=	0011
4	=	0100
5	=	0101
6	=	0110
7	=	0111
8	=	1000
9	=	1001

Die Zahl $719_{(10)}$ wird im BCD-Code wie folgt wiedergegeben:

$$\underbrace{0111}_{7}\underbrace{0001}_{1}\underbrace{1001}_{9}{}_{(BCD)}$$

Ebenfalls auch:

$581_{(10)}=010110000001_{(BCD)}$

Dabei ist zu bedenken, daß dies ein Code-System ist und keine Umrechnung vom Dezimalsystem in das Binär-System.

$581_{(10)}$ und $0101\ 10000001_{(2)}$ ist also nicht dasselbe.

3.4 Der Hexadezimal-Code

Die Anzahl verschiedener Zahlen, die in einem Register untergebracht werden kann, ist von der Registergröße und daher von der Wortlänge abhängig. Bei einer Wortlänge von zwei Bit können vier verschiedene Zahlen gebildet werden:

00
01
10
11

Somit sind 2^2 Kombinationen möglich. Eine Wortlänge von drei Bit ergibt 2^3 Kombinationsmöglichkeiten.

Ganz allgemein beträgt die Anzahl der Möglichkeiten 2^n, wobei n die Anzahl Bit eines Wortes bedeutet. Teilen wir ein Register in Vier-Bit-Wörter auf, dann sind pro Wort $2^4=16$ Kombinationen möglich.

Bei dieser angegebenen Aufteilung können wir nicht nur Dezimalzahlen in Codeform in die Register eingeben, sondern auch Zahlen aus dem Zahlensystem mit der Basis 16

(Hexadezimalsystem = Sedezimalsystem). Im Hexadezimalsystem haben wir jedoch nicht genügend Symbole aus dem Dezimalsystem, deshalb werden die sechs fehlenden durch die ersten sechs Buchstaben (A bis F) aus dem Alphabet ersetzt.

In der nachfolgenden Tabelle sind die sechzehn „Ziffern" aus diesem System mit dem dazu gehörigen Code, der ebenso wie der BCD-Code an das Binärsystem angelehnt ist, angegeben.

Hexadezimal	*Binär*	*Dezimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

In der Mikrocomputer-Technik ist eine Art Standard-Registerlänge gebräuchlich. In diese können 8-Bit-Datenwörter eingegeben werden. Ein derartiges 8-Bit-Datenwort wird mit „Byte" bezeichnet. Ein Byte kann eine Zahl von zwei Ziffern aus dem Hexadezimalsystem enthalten, z.B.

$A9_{(16)}=\underbrace{1010}_{A}\ \underbrace{1001}_{9}{}_{(2)}$

und

$7B_{(16)}=01111011_{(2)}$

Das Gleichheitszeichen „=" steht hier zu Recht. Da in diesem Beispiel mit vier Bits sämtliche Kombinationsmöglichkeiten ausgenutzt wurden, können die Zahlen aus dem Hexadezimalsystem in das binäre System durch einfaches Auffüllen des Codes durch besondere Ziffern umgerechnet werden.

Die Stellen in einem 8-Bit-Register werden mit einer Nummer angegeben, und zwar beginnend beim niedrigsten Bit mit B0, B1 usw. bis B7:

B7	B6	B5	B4	B3	B2	B1	B0

In der Zahl $10110110_{(2)}$ haben B0, B3 und B6 den Wert 0 und B1, B2, B4, B5 und B7 den Wert 1.

4 Rechnen im binären und hexadezimalen System

4.1 Binäre Addition

Die Addition im Binär-(Dual)-System ist im Prinzip einfach, da lediglich mit den Ziffern 0 und 1 gerechnet werden muß:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10

Beim letzten Additionsvorgang entsteht ein Übertrag (engl. carry).
Beispiel:

```
carry      11110
Summand     1101
Summand      111+
-----------------
Summe      10100
```

Ganz übereinstimmend mit den Regeln wird Bit für Bit addiert. Ein Übertrag wird in die Zeile „carry" eingetragen. So kann es zum Beispiel vorkommen, daß die Summe eine größere Anzahl von Bits enthalten kann als die Summanden. Im Mikrocomputer erfolgt die Berechnung in Registern, die z.B. 8-Bit-Dualzahlen enthalten können. Die Rechnung sieht dann wie folgt aus:

```
carry      00011110
Summand    00001101
Summand    00000111+
--------------------
Summe      00010100
```

Bei einer Addition kann es vorkommen, daß das Resultat, die Summe, für das Register zu groß ist.

```
  11011101
  01110010+
-----------
1 01001111
```

Im Mikrocomputer wird dafür gesorgt, daß das höchste Carry-Bit nicht verlorengeht. In einem besonderen Register wird an einer hierfür bestimmten Stelle eine „1" notiert

(C= 1, C von Carry). Ist kein Übertrag vorhanden, wird im Register eine „0" eingegeben: C= 0.

4.2 Binäre Subtraktion

Beim Subtrahieren muß mitunter „geborgt" werden (borgen ist englisch „borrow").

0 – 0 = 0	geborgt = 0
0 – 1 = 1	geborgt = 1
1 – 0 = 1	geborgt = 0
1 – 1 = 0	geborgt = 0

Beispiel:

borrow	00110000
Minuend	00110110
Subtrahend	00011100–
Differenz	00011010

Auch hier wird Bit für Bit subtrahiert. Das „Geborgte" steht in der Zeile „borrow".

4.3 Binäres Multiplizieren und Dividieren

Die Tabellen für Multiplizieren bieten uns keine große Schwierigkeiten:

0 X 0 = 0
0 X 1 = 0
1 X 0 = 0
1 X 1 = 1

Beispiel:

Multiplikand	00001101
Multiplikator	00001011X
Teilprodukte	00001101
	00011010
	00000000
	01101000+
Produkt	10001111

Ein Mikrocomputer kann nicht mehr als zwei Zahlen addieren. Im Beispiel zur Multiplikation muß eine Addition von vier Zahlen stattfinden. Es wird nun folgendermaßen verfahren:

Multiplikand	00001101
Multiplikator	00001011x
	00001101
	00011010+
	00100111
	01101000+
Produkt	10001111

Aus diesem Beispiel geht hervor: Das Teilprodukt ergibt sich daraus, daß der Multiplikand mit 1, 10 oder 100 oder . . . . abhängig vom Stellenwert, den die betreffende 1 im Multiplikator hat, zu multiplizieren ist.

Diese Multiplikation geht so vonstatten, daß der Multiplikand um soviel Stellen nach links verschoben wird, wie Stellen rechts von der betreffenden 1 des Multiplikators liegen:

00001101	und auch	00001101
00000100x		00000010x
00110100		00011010

Im letzten Beispiel wird der Multiplikand 00001101 mit $10_{(2)} = 2_{(10)}$ multipliziert.

Schiebt man eine binäre Zahl in ein Register um eine Stelle nach links, dann wird diese mit $2_{(10)}$ multipliziert und umgekehrt.

Schiebt man eine binäre Zahl in einem Register um eine Stelle nach rechts, dann wird diese durch $2_{(10)}$ dividiert.

Numerisch ist eine Division durch Null nicht zulässig. Divisionen werden darum wie folgt ausgeführt:

0 : 1 = 0 und 1 : 1 = 1

Eine Division ist eine wiederholte Subtraktion vom Divisor.

Divisor	*Dividend*	*Quotient*
00001101/	10001111/	00001011
	01101000–	
	00100111	
	00011010–	
	00001101	
	00001101–	
	00000000	

4.4 Hexadezimales Rechnen

Rechnen im Hexadezimalsystem ist wesentlich schwieriger als binäres Rechnen und uns nicht so geläufig wie Dezimalrechnen. Ist bei Dezimalrechnung eine Multiplikationstabelle von 10 x 10 = 100 Faktoren erforderlich, so sind im Hexadezimalsystem 16 x 16 = 256 Faktoren zu berücksichtigen.

Es ist daher wesentlich einfacher, hexadezimale Zahlen in ihren Binärcode umzuschreiben und dann binär zu rechnen:

	Hex	*Binär*
Addieren:	3D	00111101
	56+	01010110+
	93	10010011
Subtrahieren:	75	01110101
	4A–	01001010–
	2B	00101011
Multiplizieren:	0C	00001100
	0Dx	00001101x
	9C	00001100
		00110000+
		00111100
		01100000+
		10011100
Dividieren:	C6	0B00010010/11000110=00001011
	12:	10010000–
	0B	00110110
		00100100–
		00010010
		00010010–
		00000000

5 Rechnen im BCD-System

Die Addition von zwei Dezimalzahlen, deren Summe kleiner als 10 ist, bietet keine Schwierigkeiten:

Dezimal	*BCD*
03	00000011
04+	00000100+
07	00000111
06	00000110
03+	00000011+
09	00001001

Anders, wenn die Summe größer als 9 ist:

Dezimal	*BCD*
06	00000110
05+	00000101+
11	00001011

Da die Addition in den Registern binär erfolgt, ergibt sich folgendes Resultat:

$00001011_{(2)}$

Entsprechend der Addition 06 + 05 = 11 müßte das Ergebnis wie folgt lauten:

$00010001_{(BCD)}$

Die richtige Zahl ergibt sich jedoch daraus, wenn zu dem Ergebnis noch

$0110_{(2)} = 6_{(10)}$

addiert wird:

Dezimal	*BCD*	
06	00000110	
05+	00000101+	
11	00001011	
	00000110+	Korrektur
	00010001	

ebenso:

09	00001001
04+	00000100+
13	00001101
	00000110+ Korrektur
	00010011

Eine Korrektur muß dann erfolgen, wenn das Resultat einer (binären) Addition größer als $00001001_{(2)} = 09_{(16)}$.

Es gibt auch noch weitere Fälle, in denen eine Korrektur erforderlich ist:

Dezimal	*BCD*
09	00001001
08+	00001000+
17	00010001
	00000110+ Korrektur
	00010111

Auch hier ist das Ergebnis größer als $09_{(16)}$ und muß deshalb korrigiert werden. Ein Mikrocomputer betrachtet diese Zahl als „Tetrade" oder „Nibble" (Zusammenfassung von 4 Bits), wobei die erste Tetrade (B0 bis B3) gemäß der binären Addition einen Inhalt von $0001_{(2)}$ hat, aus dem aber nicht hervorgeht, ob eine Korrektur notwendig ist. Andererseits aber wird ein Übertrag von der ersten in die zweite Tetrade vorgenommen. Bei vielen Mikroprozessoren wird in einem besonderen Register an einer bestimmten Stelle eine 1 notiert (H = 1, H von Half carry). Ohne Übertrag ist H = 0; im vorliegenden Fall, da H = 1, ist eine Korrektur am Ergebnis vorzunehmen.

Was für die erste Tetrade gilt, gilt auch für die zweite. Auch hier muß korrigiert werden, da das Ergebnis in dieser Tetrade größer als $9_{(16)}$ ist, als ob es ein Übertrag wäre. Ein Übertrag der zweiten Tetrade wird wie beim binären Addieren als C = 1 notiert.
Beispiel:

Dezimal		*BCD*	
23		00100011	
95+		10010101+	
118	C = 0	10111000	H = 0
		01100000+	Korrektur
	C = 1	00011000	
58		01011000	
89+		10001001+	
147	C = 0	11100001	H = 1
		01100110+	Korrektur
	C = 1	01000111	

98		10011000	
89+		10001001+	
187	C = 1	00100001	H = 1
		01100110+	Korrektur
	C = 1	10000111	

Auch bei diesem letzten Beispiel ist das Resultat größer als $99_{(10)}$ (187). Es muß daher ein Übertrag erfolgen. Obgleich im Ergebnis der Addition als Korrektur C = 0 ist, muß in diesem Fall der gleiche Wert für C wie in der ersten Addition, d.h. „1" beibehalten werden (C war 1, C bleibt also 1).

Angenommen, das Resultat der hohen Tetrade einer Summe beträgt vor der Korrektur n und das von der niedrigen Tetrade m, dann ist die Zahl n, m. Für diese Zahl

B7		B4		B3			B0
1	1	0	1	0	1	1	0

ist n 1101 und m 0110. In nachfolgender Tabelle sind für die verschiedenen Möglichkeiten von n und m die Wertigkeiten der hohen und niedrigen Tetrade vor und nach Korrektur angegeben.

Vor Korrektur:

C	B7 . . . B4	H	B3 . . . B0
0	n<10	0	m<10
0	n<10	1	m<10
0	n<9	0	m>9
1	n<10	0	m<10
1	n<10	1	m<10
1	n<10	0	m>9
0	n>9	0	m<10
0	n>9	1	m<10
0	n>8	0	m>9

Nach Korrektur:

C	B7 . . . B4	H	B3 . . . B0
0	n	0	m
0	n	0	m+6
0	n+1	1	m+6
1	n+6	0	m
1	n+6	0	m+6
1	n+6+1	1	m+6
1	n+6	0	m
1	n+6	0	m+6
1	n+6+1	1	m+6

6 Negative Zahlen

6.1 Das Zweierkomplement

Aus der Algebra ist bekannt, daß Zahlen kleiner als Null negativ sind und mit dem Zeichen „–" vor der Zahl gekennzeichnet werden:

–1, –2, –3 usw.

Eine negative Zahl erhält man, wenn eine Zahl von einer kleineren zu subtrahieren ist:

```
 3
 9–
 —
–6
```

Eine solche Kennzeichnung negativer Zahlen kennt der Mikrocomputer nicht. Wird binär eine Zahl von einer kleineren Zahl subtrahiert, kann festgestellt werden, wie eine negative Zahl dann im Ergebnisregister aussieht:

Dezimal		*binär*	*hex*
03	C=1	00000011	03
09–		00001001	09–
–06		11111010	FA

Durch die Zahl $11111010_{(2)}$ wird deutlich, daß dies der negative Wert von $0000110_{(2)}$ (06_{10}) ist. Dasselbe Ergebnis erhält man durch Subtraktion von 06 von 00:

Dezimal		*binär*	*hex*
00	C=1	00000000	00
06–		00000110–	06–
–06		11111010	FA

Die Zahl $11111010_{(2)}$ ist das Auffüllen (Komplement) von $00000110_{(2)}$ bis 100000000, also:

```
        11111010
        00000110+
        --------
C=1     00000000
```

Da $100000000_{(2)} = 2^8{}_{(10)}$ ist, spricht man also vom Zweierkomplement. Es ist dies also eine zweite Art, um das Zweierkomplement von einer Zahl zu finden:

	100000000
	1–
	11111111
Zahl	00000110–
Zwischenresultat	11111001
	1+
Zweierkomplement	11111010

Es ist darauf zu achten, daß alle Bits vom Zwischenergebnis umgekehrt zu dieser Zahl anzugeben sind. Das Zwischenergebnis ist die Inversion der Zahl. Das Zweierkomplement einer binären Zahl ergibt sich durch Inversion und Addition von 1.

Beispiel:

Zahl	00101101
Inversion	11010010
	1+
Zweierkomplement	11010011

6.2 Rechnen im Zweierkomplement-System

Der Mikrocomputer benötigt das Zweierkomplement zur Subtraktion von zwei Zahlen. Vom Subtrahend wird das Zweierkomplement definiert und dann der Minuend addiert:

Dezimal		*binär*	
08		00001000	
06–		11111010+	Zweierkomplement von $06_{(10)}$
02	C=1	00000010	
04		00000100	
06–		11111010+	
–02	C=0	11111110	Zweierkomplement von $02_{(10)}$

Im letzten Beispiel ist das Ergebnis negativ. Besonders wichtig ist es, an einer Zahl zu erkennen, ob sie positiv oder negativ ist. Um dies zu verwirklichen, wird nur die eine Hälfte des Zahlenbereichs für die positiven Zahlen benutzt. Die andere Hälfte ist dann für die negativen Zahlen reserviert.

binär	*hex.*
00000000–01111111	00–7F

Die anderen 128 Kombinationen sind für die Zahlen −128 bis −1 bestimmt.

binär	*hex.*
10000000–11111111	80–FF

Die negativen und positiven Zahlen sind an Bit 7 zu erkennen.

Bei einer positiven Zahl ist Bit 7 „0",
bei einer negativen Zahl ist Bit 7 „1".

Nun ist wieder eine Stelle im „Sonderregister" vorgesehen (das Status-Register), worin das Ergebniszeichen festgelegt wird:

Ist das Resultat negativ (B7 = 1), dann wird eine 1 notiert (N = 1)
Ist das Resultat positiv (B7 = 0), dann wird eine 0 notiert (N = 0).

Bei der Addition von zwei positiven Zahlen kann es allerdings vorkommen, daß Bit 7 „1" wird:

```
Summand   01101101
Summand   01011010+
          --------
Summe     11000111
```

Der Computer stellt fest, daß das Resultat negativ ist (N = 1); das ist falsch und darum ist in dem Statusregister noch ein Zeichen V vorgesehen (V von overflow).
Der Inhalt von V wird nur dann „1", wenn der Inhalt von N nicht richtig ist.

V wird „1", wenn beim Addieren von zwei positiven Zahlen N „1" wird,
V wird „1", wenn beim Addieren von zwei negativen Zahlen N „0" wird.

Im nachfolgenden Beispiel werden zwei negative Zahlen addiert. Hierbei wird im Resultat Bit 7 zu „0":

```
          10010010
          10101110+
          --------
C = 1     01000000    N = 0    V = 1
```

Wir haben uns mit C, H, N und V beschäftigt, die sämtlich im Statusregister enthalten sind. Im nachfolgenden Beispiel werden diese Zeichen an Hand einer Addition von zwei positiven Zahlen erläutert:

```
01011001      C = 0
01001010+     H = 1
--------
10100011      N = 1
              V = 1
```

Die Subtraktion von zwei dezimalen Zahlen erfolgt am besten mit dem Komplementsystem.

Beispiel:

```
 47
 15–
----
 32
```

Das Zehner-Komplement von 15 (Auffüllen auf 10^2) ergibt sich so:

```
100
 15–
----
 85
```

Addieren:

Dezimal		*BCD*	
47		01000111	
85+		10000101+	
132	C=0	11001100	H = 0
		01100110+	Korrektur
	C=1	00110010	

7 Logisches Arbeiten

7.1 Logische Funktionen UND, ODER, EXCEPT, NICHT

Außer mathematischen Aufgaben kann die Zentraleinheit auch logische Befehle ausführen. Dem Mikroprozessor können vier logische Vorgänge zugemutet werden:

„UND" (and), Symbol $\wedge$
„ODER" (or), Symbol $\vee$
„EXCEPT", Symbol $\forall$
„NICHT", Symbol $\bar{}$ ($\bar{a}$ = NICHT a).

Beim Aufstellen von Arbeitstabellen kann von bestimmten Entscheidungen ausgegangen werden. Diese Entscheidungen oder Vorschläge müssen kontrollierbar sein, ob „richtig" oder „falsch". Eine brauchbare Entscheidung kann sein, „es regnet", aber nicht „es ist kalt". Die letzte ist z. B. eine subjektive Erfahrung. Das „richtig" oder „falsch" einer Entscheidung kann mit „1" oder „0" gekennzeichnet werden. Die Entscheidung wird mit einem Buchstaben angegeben. Dieser Buchstabe ist dann die „Veränderliche", z.B.

es regnet = **x**, **x** ist die Veränderliche
es regnet ist richtig → **x** = 1
es regnet ist falsch → **x** = 0
x ist veränderlich über {0, 1}.

Zwei Entscheidungen werden durch ein „Bindeglied" sinngemäß verbunden:

es regnet „und" es hagelt
es regnet ist **x**,
es hagelt ist y,
das Bindeglied ist „und".

Ob die Entscheidung richtig oder falsch ist, hängt vom Bindeglied und von der Richtigkeit der speziellen Entscheidung ab. Folgende Tabelle kann aufgestellt werden:

x	*y*	*x*∧*y*	
0	0	0	x = 0 es regnet ist falsch
0	1	0	x = 1 es regnet ist richtig
1	0	0	y = 0 es hagelt ist falsch
1	1	1	y = 1 es hagelt ist richtig

Bei zwei Veränderlichen, die beide 0 oder 1 sein können, finden wir vier Möglichkeiten ($2^2 = 4$):

1. Regel x = 0 y = 0 $x \wedge y = 0$
2. Regel x = 0 y = 1 $x \wedge y = 0$
3. Regel x = 1 y = 0 $x \wedge y = 0$
4. Regel x = 1 y = 1 $x \wedge y = 1$

Das Ergebnis ist nur dann 1, wenn beide Veränderlichen 1 sind.

Diese Tabelle ist mit den Entscheidungen zu kontrollieren. Selbstverständlich ist die Entscheidung „es regnet" und „es hagelt" nur dann richtig, wenn es gleichzeitig regnet und hagelt. Die Tabelle wird Wahrheitstabelle genannt.

Weniger selbstverständlich ist die Richtigkeit der Tabelle für das Bindeglied „oder":

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Das Resultat ist nur dann Null, wenn beide Veränderlichen Null sind.

In der Umgangssprache ist es richtig, wenn die Entscheidung „es regnet oder es hagelt" richtig ist, wenn eine der beiden Entscheidungen richtig ist (Regel 2 und 3). Im allgemeinen wird die 4. Regel als nicht richtig angesehen. Dennoch ist diese Tabelle logischerweise entsprechend definiert „einschließlich" Regel 4. Gebrauchen wir das Bindeglied „exclusiv oder", dann ist die gesamte Entscheidung richtig, sofern eine der beiden richtig ist, deshalb ohne (exclusiv) Regel 4:

x	y	$x \forall y$
0	0	0
0	1	1
1	0	1
1	1	0

Das Resultat ist nur dann Null, wenn beide Veränderlichen „gleich" sind.

Die vierte Funktion ist die „NICHT"-Funktion: (Negation) Wenn die Entscheidung „es regnet" richtig ist, dann ist die Entscheidung „es regnet nicht" falsch.

es regnet	x
es regnet nicht	$\bar{x}$

Es gibt also nur zwei Möglichkeiten:

x	$\bar{x}$
0	1
1	0

Die Tabelle entspricht ganz der Definition. x und $\bar{x}$ sind untereinander „invertiert".

7.2 Anwendungen

Da der Mikrocomputer binär arbeitet, können auch logische Verknüpfungen von der Zentraleinheit verarbeitet werden. Die Funktionen werden in zwei Wörtern gleicher Länge geschrieben, die dann Bit für Bit das Ergebnis bestimmen (0 oder 1):

$$\begin{array}{rl} a & 01101011 \\ b & \underline{10110001} \\ a \wedge b & 00100001 \end{array}$$

$$\begin{array}{rl} a & 01101011 \\ b & \underline{10110001} \\ a \vee b & 11111011 \end{array}$$

$$\begin{array}{rl} a & 01101011 \\ b & \underline{10110001} \\ a \forall b & 11011010 \end{array}$$

$$\begin{array}{rl} a & 01101011 \\ \bar{a} & 10010100 \end{array}$$

Als Beispiel folgt hier eine Methode zur Wertbestimmung eines definierten Bits aus einem Wort. Angenommen, es soll aus dem Wort

01001010 (Zahl a)

der Wert von B_2 ermittelt werden. Hierzu muß eine UND-Verknüpfung durchgeführt werden, und zwar mit der Zahl 00000100, eine sogenannte „Maskierung" (Maske).

$$\begin{array}{lrl} & a & 01001010 \\ \text{Maske} & b & \underline{00000100} \\ & a \wedge b & 00000000 \end{array}$$

Wir sehen daraus, daß das betreffende Bit „0" ist, denn das Ergebnis ist Null. Es ist darauf zu achten, daß, wenn der Wert des dritten Bit einer Zahl bestimmt werden soll, auch das einzige Bit mit „1" in der Maskierung tatsächlich das dritte Bit ist. Im Statusregister ist nun ein Bit vorgesehen, womit dieses Ergebnis gekennzeichnet wird, nämlich Z (Z von Zero):

Z = 1, wenn das Resultat eines Vorganges 0 ist, sonst wird Z Null.

$$\begin{array}{lrll} & a & 01001110 & \\ \text{Maske} & b & \underline{00000100} & \\ & a \wedge b & 00000100 & Z = 0 \end{array}$$

Das dritte Bit der Zahl a ist nunmehr „1". Weiterhin ergibt sich die Möglichkeit, den Inhalt einer Tetrade zu isolieren:

		binär	*hex*
	a	01001110	4E
Maske	b	00001111	0F
	a∧b	00001110	0E

Dadurch, daß die UND-Verknüpfung 00001111 zur Anwendung kommt, wird im Ergebnis die zweite Tetrade Null, und der Wert des Ergebnisses ist gleich dem der ersten Tetrade der Zahl a.

Durch die Kombination einer UND- und einer ODER-Verknüpfung kann die erste Tetrade einer Zahl a mit der zweiten Tetrade einer Zahl b zu einer neuen Zahl c kombiniert werden:

Zahl a: $9B_{(16)}$ Zahl b: $A7_{(16)}$ Zahl c: $AB_{(16)}$

	binär	*hex*		*binär*	*hex*
a	10011011	9B	b	10100111	A7
p	00001111	0F	q	11110000	F0
a∧p=r	00001011	0B	b∧q=s	10100000	A0

	binär	*hex*
r	00001011	0B
s	10100000	A0
rVs= c	10101011	AB

Zuerst wird der Wert der ersten Tetrade von Zahl a, dann der Wert der zweiten Tetrade der Zahl b bestimmt und das Ergebnis durch eine ODER-Verknüpfung gefunden.

Die Exklusiv-ODER-Verknüpfung bietet die Möglichkeit, das Zweierkomplement einer Zahl zu ermitteln. Die Zahl, von der das Zweierkomplement bestimmt werden soll, wird in „EX-OR"-Verknüpfung mit 11111111 als Maske versehen.

	a	01101011
Maske	b	11111111
a∀b=	$\bar{a}$	10010100

Dieses Resultat stellt nunmehr die Invertierung der Zahl a dar. Zur Ermittlung des Zweierkomplementes der Zahl a muß zur Inversion noch 00000001 addiert werden.

	$\bar{a}$	10010100
		00000001+
Zweierkomplement von	a	10010101.

8 Das Programm

8.1 Das Flußdiagramm

Die Zentraleinheit eines Mikrocomputers ist nicht in der Lage, mehr zu leisten, als ihr aufgetragen wird. Ein Auftrag soll bzw. kann sein:

Addiere die Zahl a zur Zahl b;
Subtrahiere die Zahl b von der Zahl a;
Schiebe die Bits von Zahl a eine Stelle nach rechts, usw.

Jeder Auftrag wird als Instruktion bezeichnet, und mit Hilfe einer Anzahl dieser Instruktionen, die nacheinander ausgeführt werden, kann der Computer eine bestimmte Berechnung durchführen, z.B.:

1. Addiere Zahl a zu Zahl b,
2. subtrahiere von dem Resultat die Zahl c,
3. dividiere den erhaltenen Wert durch zwei.

So verläuft die Berechnung von $\frac{a+b-c}{2}$

Dieses Verfahren mit Instruktionen, die zu dem gewünschten Resultat führen, wird Algorithmus oder Programm genannt.

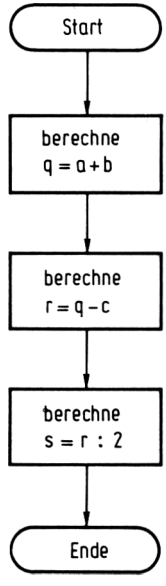


Abb. 3 Detailliertes Flußdiagramm

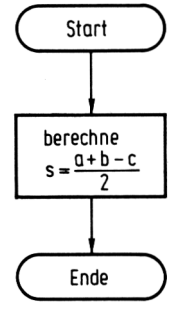


Abb. 4 Grobes Flußdiagramm der gleichen Aufgabe

Dieses Programm muß der Zentraleinheit angeboten werden. Dazu wird es in richtiger Reihenfolge in einen Speicher übertragen. Dieser Speicher besitzt eine Anzahl von Plätzen, in denen die Instruktionen gespeichert sind. Beim Abarbeiten eines Programmes sieht der Computer nach, welche Instruktion im Speicher steht, beginnt dabei selbstverständlich mit der ersten, führt diese Instruktion aus, sieht nach der nächsten Instruktion usw. Auch die Daten (die Zahlen a, b und c) findet er auf bestimmten Plätzen im Speicher.

Das erwähnte Programm wird durch ein Blockschema wie in *Abb. 3* verdeutlicht. Dieses Blockschema wird als Flußdiagramm bezeichnet (Flowchart). In diesem Schema wird jede Instruktion mit einem Block dargestellt, in dem die Art der Instruktion angegeben ist. Die Pfeile in den Verbindungen zwischen den Blöcken geben die „Stromrichtung" an; daraus ergibt sich die Reihenfolge, die das Programm durchlaufen muß. Es ist nicht unbedingt erforderlich, daß in jedem Block nur eine Instruktion steht. Es ist auch möglich, in einem Block einen Programmteil oder ein ganzes Programm wiederzugeben, wie dies in *Abb. 4* dargestellt ist.

8.2 Sprungbefehle

Der Speicher eines Mikrocomputers besteht meist aus Registern von jeweils acht Bit, und jedes Register ist ein Speicherplatz. Das bedeutet, daß ein Befehl „codiert" in den Speicher eingebracht werden muß. Der Speicher wird mit den Instruktionen „geladen". Auch das Laden des Speichers läuft über die Zentraleinheit.

Unterstellen wir, daß für ein bestimmtes Programm fünf Speicherplätze mit $00000000_{(2)} = 00_{(16)}$ geladen werden sollen. Die Speicherplätze sind von 1 bis 5 numeriert (sp1, sp2 . . . sp5). Das Programm kann gemäß Abbildung 5 ablaufen. Das Zeichen $ für eine Zahl gibt an, daß es sich um eine Zahl aus dem Hexadezimalsystem handelt. „Lade sp1 mit $ 00" wird hier geschrieben:

$ 00 → sp1 ($ 00 nach sp1).

Das aufgezeigte Programm kennt fünf Instruktionen, es geht aber auch kürzer.

Im Programm nach *Abb. 6* wird ein bestimmtes Register (z.B. ein Speicherplatz) mit x angegeben und mit $ 01 geladen.

Dieses Register bezeichnet man mit „Zähler". Mit dem nun folgenden Befehl wird sp1 mit $ 00 geladen und der Zähler um 1 erhöht. Das Programm springt auf den Befehl mit der Marke (label) „loop" zurück. Dieser Sprungbefehl („jump") muß erteilt werden, da der Computer von sich aus nichts unternimmt. Der nächste Befehl lädt sp2 mit $ 00, also x = 2.

Dieser Kreislauf setzt sich durch Aufladen der nächsten Speicherstellen fort. Der Computer kommt jedoch noch immer nicht zur Ruhe, denn, sobald x angehoben wird, wird die Schleife durchlaufen und der nächstfolgende Speicherplatz geladen.

Hier erfolgt kein gewöhnlicher Sprung (entsprechend der Reihenfolge des Programms), sondern ein „bedingter" Sprung.

Ein bedingter Sprungbefehl (branch) deutet darauf hin, daß ein Sprung (hier zurück auf „loop") nur unter einer bestimmten Bedingung erfolgt.

Im Programm nach *Abb. 7* wird der Zähler mit $ 05 geladen. Der erste Speicherplatz, der nunmehr geladen wird, ist sp5 (x = 5). Der Zähler wird um 1 vermindert. Der nächste Block (Raute) stellt die Bedingung: wenn x = 0, dann erfolgt kein Sprung. Soweit ist es

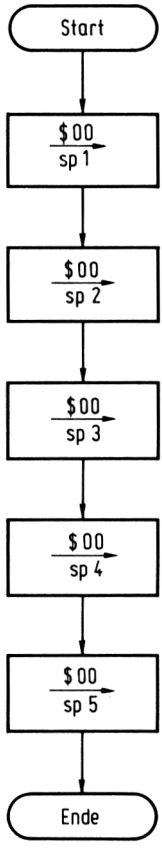


Abb. 5 Flußdiagramm zum Laden von fünf Speicherplätzen mit Null

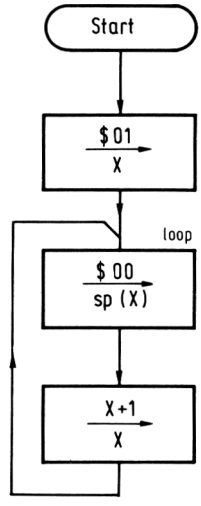


Abb. 6 Programmschleife mit unbedingtem Sprung

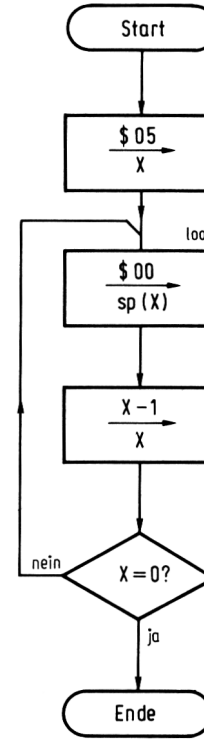


Abb. 7 Laden von fünf Speicherplätzen unter Zuhilfenahme eines bedingten Sprungbefehls

aber noch nicht. Das Programm läuft zurück auf „loop", sp4 wird geladen und so weiter, bis x = 1 wird. Nach dem Laden des letzten Speicherplatzes (sp1) wird der Zähler wieder um 1 vermindert und steht jetzt auf 0. Ein Rücksprung auf „loop" erfolgt nicht, das Programm ist somit abgelaufen.

Außerdem spielt Z (das Null-Flag) im Statusregister eine Rolle. Steht nach der Verminderung der Zähler auf 0, wird das Statuszeichen Z = 1 gesetzt.

8.3 Programme für logische Funktionen

Die Bedingung x = 0 im Programm nach Abb. 7 kann auch als eine Entscheidung über richtig (ja) oder falsch (nein) aufgefaßt werden. In *Abb. 8* ist ein Unterprogramm mit 3 Registern wiedergegeben. Von diesem Unterprogramm ist eine Wahrheitstabelle aufzustellen, in der eine „richtige Behauptung" mit 1 und eine „falsche" mit 0 bezeichnet sind. 1 oder 0 haben hier keinen Bezug auf den tatsächlichen Inhalt der Register x oder y, sondern auf das „alles oder nichts" der Entscheidung über den Registerinhalt. a ist die Entscheidung x = 0, b die Entscheidung y = 0, die Spalte c gibt zum Unterschied zum Vorher-

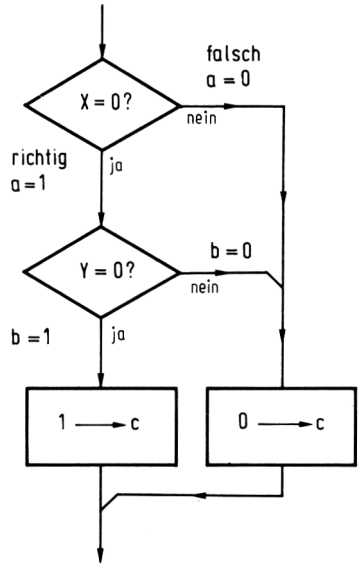


Abb. 8 Entscheidungen abhängig von Registerinhalten: UND

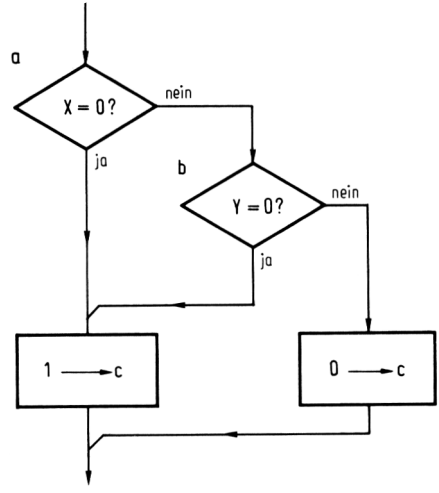


Abb. 9 Ein weiteres Beispiel für logische Entscheidungen: ODER

gehenden den wahren Inhalt des Registers an. Darum ist hier nicht die Rede von einer Entscheidung, sondern von einem Ergebnis.

a	*b*	*c*
0	0	0
0	1	0
1	0	0
1	1	1

Diese Tabelle gilt für die Funktion c = a∧b

Auch für das Unterprogramm nach *Abb. 9* wird eine Wahrheitstabelle aufgestellt:

a	*b*	*c*
0	0	0
0	1	1
1	0	1
1	1	1

Diese Tabelle gilt für die Funktion c = a∨b, so daß mit dem Unterprogramm nach Abb. 9 eine „ODER"-Funktion zu verwirklichen ist.

Weiterhin ist auch die Realisierung der Funktion c = a∀b möglich. Hierzu dient das Unterprogramm in *Abb. 10*. In dieser Abbildung wird auf die beiden Kreise mit einer Ziffer hingewiesen: (1)

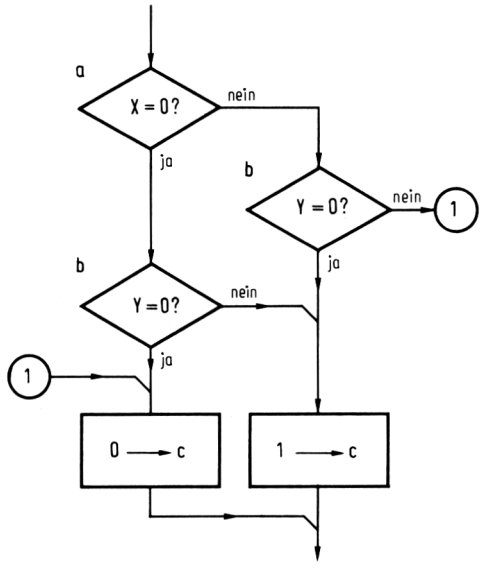


Abb. 10 Exklusiv-Oder-Verknüpfung von Entscheidungen

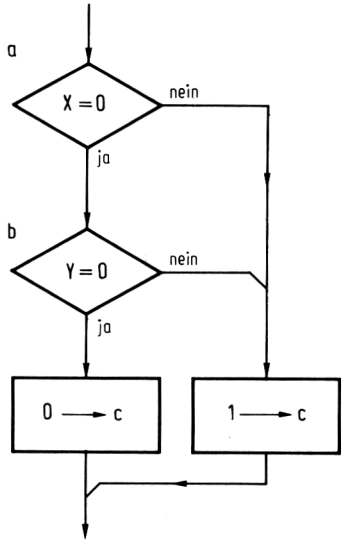
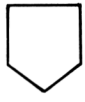


Abb. 11 NAND-Verknüpfung bei logischen Entscheidungen

Zwischen den gleichnumerierten Punkten befindet sich eine Verbindung. Diese Art der Darstellung vermeidet Unklarheiten infolge sich kreuzender Leitungen.

Dehnt sich ein Flußdiagramm über mehrere Seiten aus, dann werden die Verbindungen zwischen den verschiedenen Programmteilen und den Seiten durch folgendes Zeichen angegeben:



Die Wahrheitstabelle für Abb. 10 lautet:

a	*b*	*c*
0	0	0
0	1	1
1	0	1
1	1	0

Auch „NICHT"-Funktionen können dargestellt werden. In Abb. 11 ist das Programm für die Funktion $c = a \overline{\wedge} b$ aufgestellt.
Die Wahrheitstabelle für *Abb. 11* lautet:

a	*b*	$a \overline{\wedge} b$	*c*
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

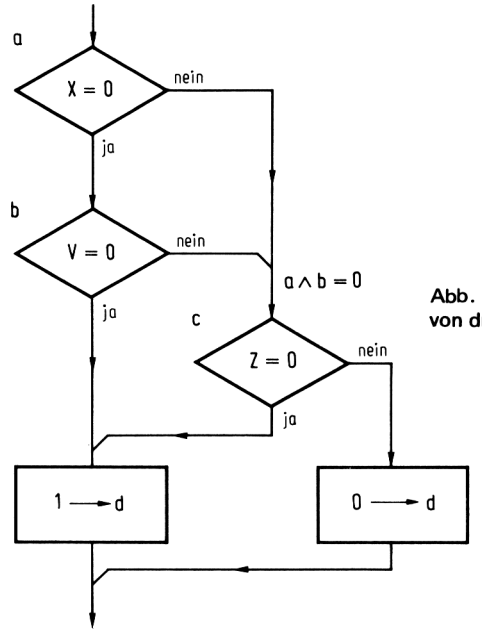


Abb. 12 Entscheidung abhängig von drei Eingangsgrößen

Es ist darauf zu achten, daß der einzige Unterschied zu Abb. 8 in der Vertauschung des Registerinhaltes c liegt. *Abb. 12* gibt die Auflösung einer zusammengefaßten Funktion wieder:

d = (a∧b)∨c

a	*b*	*c*	*a∧b*	*d*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Die Wahrheitstabelle für dieses Unterprogramm enthält drei Veränderliche, woraus sich $2^3 = 8$ Möglichkeiten ergeben. Zuerst wird die Tabelle für a∧b aufgestellt. Danach eine ODER-Funktion zwischen diesen beiden und schließlich die Tabelle für c.
Bei diesem und auch bei den vorhergehenden Beispielen sind immer die Entscheidungen x = 0, y = 0 usw. getroffen worden. Für einen bedingten Sprung kann jedoch jede andere Entscheidung eingesetzt werden, wie

x = positiv;	carry = 0;
x = negativ;	carry = 1;
x = n (n ist eine Zahl);	usw.

8.4 Der erste Programmansatz

Einleitend wurde bereits festgestellt: „Von einem Computer können nicht nur komplizierte Berechnungen durchgeführt werden, es können ihm auch Probleme angeboten werden, die durch den Menschen nicht ohne weiteres zu lösen sind, da hierfür spezielle Rechenmethoden erforderlich sind."

Daraus darf jedoch nicht geschlossen werden, daß ein Computer selbständig in der Lage ist, Probleme zu lösen. Die Auflösung von einem Problem muß dem Computer erst durch den Menschen in Form eines Programmes eingegeben werden. Erst dann ist das Problem mit Hilfe der computereigenen Möglichkeiten lösbar, was dem Menschen ohne Computer nicht möglich gewesen wäre.

Vor einer Programmaufstellung ist eine grundlegende Durchleuchtung des zu lösenden Problems erforderlich, und dazu eine entsprechende Analyse, nach der dann eine Lösungsmöglichkeit gesucht werden kann, unter Berücksichtigung der Möglichkeiten des zur Verfügung stehenden Computers. Diese hängen von der Computertype ab. Ein für alle Computer geeignetes Programm aufzustellen, ist daher nicht durchführbar.

Wir sollten uns nun mit der Lösung des „Artillerieproblems" nach Abb. 1 befassen. Es handelte sich darum, aus v.t den Abstand MT zu berechnen, wobei v die Fluggeschwindigkeit des Flugkörpers und t die Zeit, die das abgeschossene Projektil (Geschoß) für die Flugbahn 0T benötigt, bedeuten. Der Punkt T ist der Auftreffpunkt, wo sich Flugkörper und Projektil treffen sollen (*Abb. 13*).

Aus $t = f(\vec{a_t})$ folgt:

$$\vec{a_t} = \overrightarrow{a_m} + \overrightarrow{v \cdot f(a_t)}$$

Die Zeit, die das Projektil für den Flug über den Weg $OT = \vec{a_t}$ benötigt, ist jedoch mit diesem nicht proportional, sondern eine Funktion von $\vec{a_t}$, daher $t = f(\vec{a_t})$. Da der genaue Abstand nicht bekannt ist, läßt sich t nur als Funktion eines bekannten Weges, nämlich $\vec{a_m}$ berechnen. Hieraus ergibt sich:

$$\vec{a_1} = \overrightarrow{a_m} + \overrightarrow{v \cdot f(a_m)} \qquad (1)$$

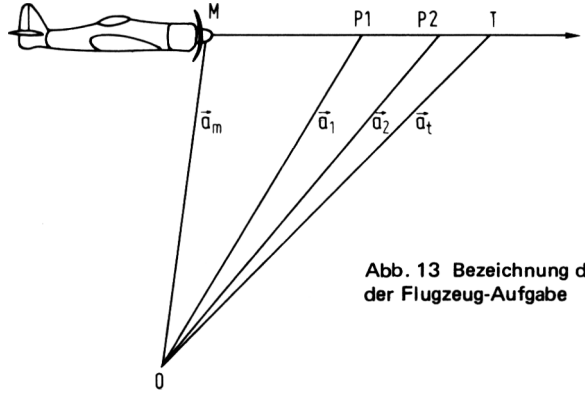


Abb. 13 Bezeichnung der Wegstrecken bei der Flugzeug-Aufgabe

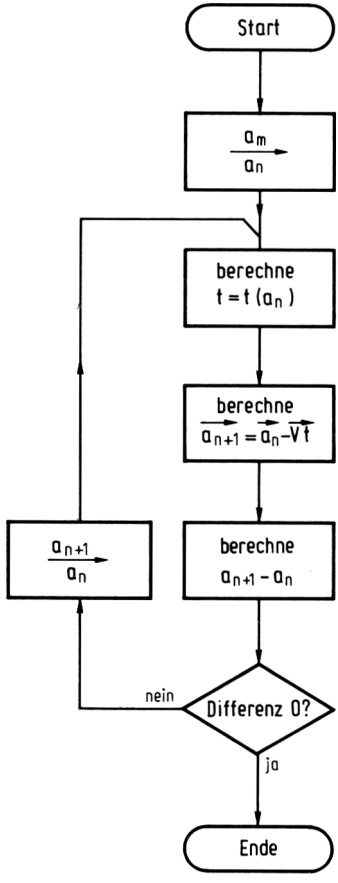


Abb. 14 Flußdiagramm zum iterativen Errechnen der Flugbahn

Der nächste Schritt ist die Berechnung von t als Funktion des ermittelten Weges $\vec{a}_1$.

$$\vec{a}_2 = \vec{a}_m + \overrightarrow{v \cdot f(a_1)}$$

Die Berechnung wird nun laufend mit dem immer wieder neu gefundenen Abstand wiederholt.

Je mehr Berechnungen ausgeführt werden, umso mehr nähert sich der ermittelte Abstand dem Wert a_t, wobei die Differenz zwischen dem letzten und dem vorletzten berechneten Abstand ständig geringer wird. Nähert sich die Differenz dem Wert Null, dann ist a_t gefunden.

Es ergibt sich folgendes Flußdiagramm (*Abb. 14*): Zuerst wird ein Register mit der Bezeichnung a_n mit dem gegebenen Abstand $\vec{a}_m$ geladen. Damit werden t und anschliessend $\vec{a}_1 = \overrightarrow{a_{n+1}}$ (Formel 1) berechnet. Die Differenz zwischen a_{n+1} und a_n ist jedoch noch nicht Null, die Rechnung wird mit dem neuen Abstand wiederholt, bis a_2 ermittelt ist, dann nochmals wiederholt usw., bis zum endgültigen Wert a_t.

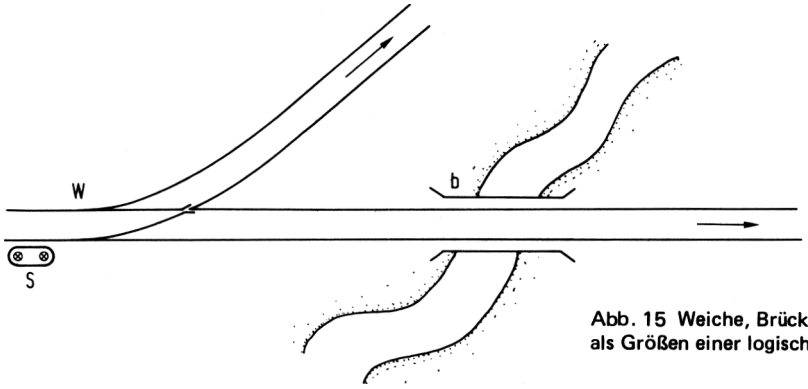


Abb. 15 Weiche, Brücke und Signal als Größen einer logischen Gleichung

Dieses Flußdiagramm gibt das Programm nur in groben Zügen wieder, wie es bei größeren Programmen üblich ist. In speziellen Flußdiagrammen müssen die Blöcke weiter verfeinert werden. Dieses Beispiel ist der erste Ansatz für eine Programmentwicklung.

Ein Beispiel für die Lösung einer logischen Gleichung finden wir bei einer Modelleisenbahn. In *Abb. 15* ist ein Gleis mit einer Weiche, einer Brücke und einem Signal dargestellt. Die Weichenlage wird als binäre Zahl ausgedrückt:

links abbiegen 1
geradeaus 0

Ebenfalls auch der Stand der Brücke:
geschlossen 1
offen 0

Das Signal hat ein rotes und ein grünes Licht:
grün 1
rot 0

Die Signalfarbe ist von der Weichenlage und dem Stand der Brücke abhängig. Hier wird eine Wahrheitstabelle aufgestellt:

w	*b*	*s*
0	0	0
0	1	1
1	0	1
1	1	1

Das Ergebnis aus der Wahrheitstabelle ist s = wVb.
Die Zahlen, die Stand von Weiche und Brücke darstellen, werden in zwei Register, mit w und b gekennzeichnet, eingegeben. Diese Register werden im Programm (*Abb. 16*) zur Darstellung der ODER-Funktion benötigt. Die Wahrheitstabelle ist an Hand des tatsächlichen Inhaltes der Register aufgestellt (w = 0 heißt geradeaus). Die Beurteilung muß eine richtige oder eine falsche Entscheidung ergeben.

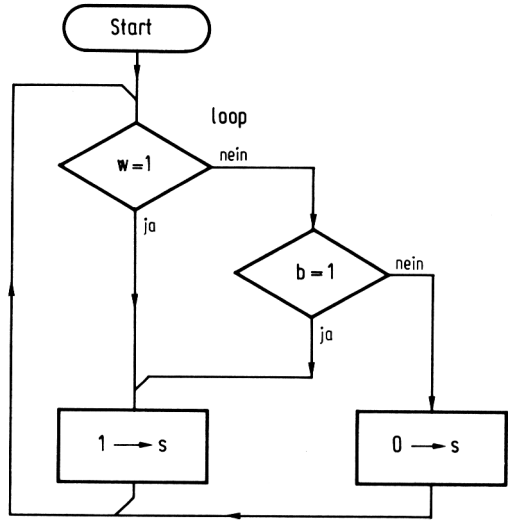


Abb. 16 Flußdiagramm zum Stellen des Signals abhängig von Weiche und Brücke

Da in den Spalten w oder b eine Null steht (und daher der Stand der Weiche oder der Brücke Null ist), ergibt sich eine falsche Entscheidung und deshalb w = 1 oder b = 1.

Der Inhalt des Registers s gibt an, ob das Signal rot (0) oder grün (1) sein muß.

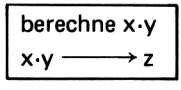
Der Stand von Brücke oder Weiche muß ständig überwacht werden. Das Programm springt daher ständig nach „loop" zurück (unbedingter Sprung), so daß der Kreis ständig neu durchlaufen wird und der Inhalt von s fortwährend vom Inhalt von w und b in Abhängigkeit gebracht wird.

8.5 Unterprogramme

Es soll ein Programm für folgende Gleichung aufgestellt werden:

$e = (a \cdot b : c)\, d$

Dazu werden drei Register benötigt, die mit x, y und z bezeichnet werden. Das Flußdiagramm ist in *Abb. 17* dargestellt. Das Ergebnis e finden wir nach Abarbeiten des Programms im Register z. Hierzu muß zweimal dasselbe Programm durchlaufen werden:



Im allgemeinen kennt die Zentraleinheit keinen Multiplikationsbefehl. Hierfür ist ein besonderes Programm erforderlich. Um zu verhindern, daß in einem Programm mehrere gleiche Teile auftreten (wie in Abb. 17), wird ein Programmteil als „Subroutine" (Unterprogramm) ausgeführt. Eine Subroutine kann als selbständiges Programm nach Belieben

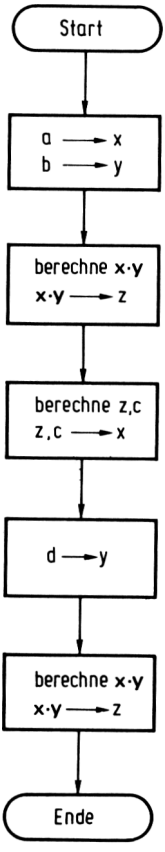


Abb. 17 Flußdiagramm zur Lösung einer mathematischen Gleichung

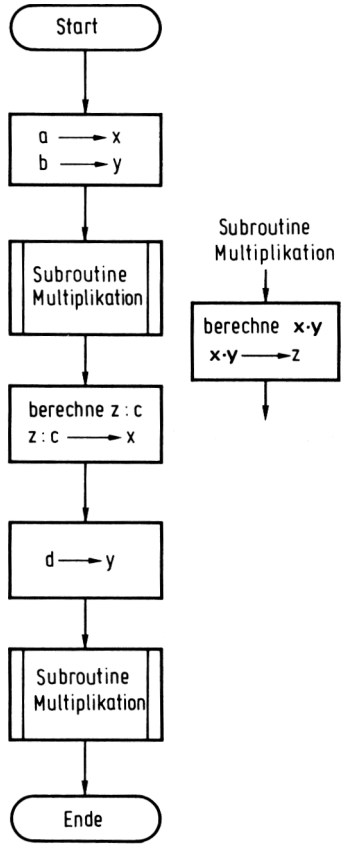


Abb. 18 Verwendung eines Unterprogramms, das mehrmals vom Hauptprogramm aufgerufen werden kann

auch in andere Progrramme eingesetzt und mit Sprungbefehlen beliebig oft aufgerufen werden.

Das Programm läuft nun wie folgt ab (*Abb. 18*): Zuerst werden die Zahlen a und b in die Register x bzw. y geladen. Nun wird auf die Subroutine gesprungen (man sagt auch, die Subroutine wird „aufgerufen"), hier das Produkt der Zahlen x und y berechnet und das Resultat in x geladen. Am Ende der Subroutine steht der Befehl: „Zurück zum Hauptprogramm". Dies ist ein Sprungauftrag und wir sind wieder an der Stelle angekommen, wo das Hauptprogramm verlassen wurde. Nun wird z:c berechnet und in x geladen. Die Zahl d wird in das Register y geladen und dann wieder auf die Subroutine gesprungen. Nach der Berechnung von x · y und nachdem das Resultat in z angegeben wurde, wird auf das Hauptprogramm zurückgesprungen, und das Programm ist abgelaufen.

9 Der Mikrocomputer

9.1 Die wichtigsten Bauelemente

Das Blockschema eines Computersystems ist aus *Abb. 19* ersichtlich. Hierin sind eine Anzahl wichtiger Bauelemente zu erkennen, nämlich Datenbus, Adressenbus, Zentraleinheit, Speicher, Ein/Ausgabe, Interface und periphere Funktionseinheiten. Innerhalb dieses Computersystems befindet sich der eigentliche Computer mit Ein/Ausgabe, Speicher, Zentraleinheit, Datenbus und Adressenbus.

Durch den Anwender müssen dem Computer Informationen angeboten werden. Diese werden „Daten" genannt und bilden die Information für den Computer. Nachdem der Computer diese Daten verarbeitet hat, wird das gewünschte Ergebnis wieder dem Anwender angeboten. Ein Computer kann darum auch als ein „informations-verarbeitendes System" bezeichnet werden. Die Informationen, die Daten also, bestehen innerhalb des Computers aus binären Zahlen, deren Wortlänge von dem betreffenden Computer abhängen. Im allgemeinen ist die Anzahl Bits/Wort ein Vielfaches von vier. Für den Datentransport innerhalb des Computers müssen verschiedene Bauelemente verbunden werden, und zwar durch eine Anzahl Leitungen, die zusammen „Bus" genannt werden. Der Bus für den Datentransport wird deshalb auch mit „Datenbus" bezeichnet. Die Anzahl der Leitungen, aus denen der Bus besteht, hängt von der Wortlänge ab. Für je ein Bit ist eine Leitung erforderlich. Sämtliche Bits werden gleichzeitig übertragen. Man spricht daher auch von „Bit-parallelem" Datenverkehr.

Alle wesentlichen Bausteine eines Computers sind mit Registern versehen, in denen unter anderem die Daten aufbewahrt sind. Speicher und Ein/Ausgabe-Einheit werden jeweils mit einer Nummer angegeben, der sogenannten Adresse. Um anzugeben, welcher

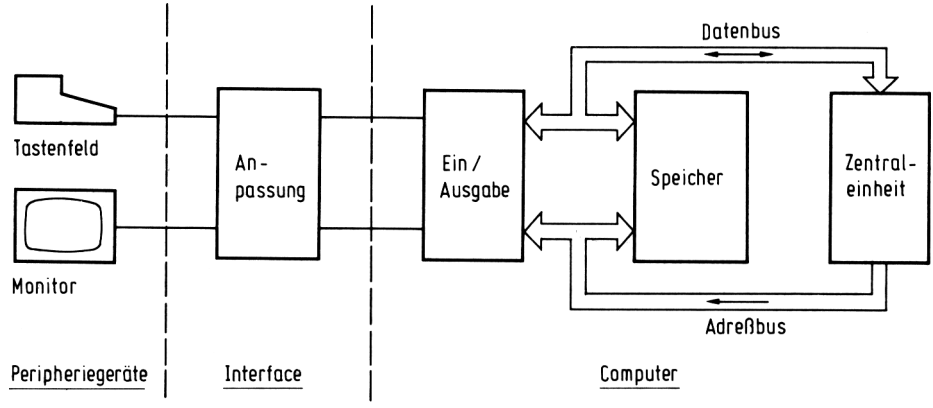


Abb. 19 Aufbau eines Mikrocomputersystems

Speicherplatz mit Daten geladen werden soll, ist ein Adressenbus erforderlich. Die Anzahl Speicherplätze, die adressiert werden können, ist von der Anzahl der Leitungen abhängig, aus denen der Bus besteht. Jede Leitung kann nur ein Bit eines Wortes erfassen, das die Adresse angibt. Bei acht Leitern können somit $2^8 = 256$ Plätze und bei 16 Leitern $2^{16} = 65536$ adressiert werden. Der Datenbus muß den Datenverkehr in zwei Richtungen abwickeln, z.B. vom Speicher zur Zentraleinheit und umgekehrt. Man spricht daher von einem bidirektionalen Datenbus. Das Adressieren erfolgt immer von der Zentraleinheit aus; die Übertragung einer Adresse im Adressenbus verläuft daher nur in einer Richtung.

Dazu kommt noch eine Anzahl weiterer Steuerleitungen, über die Befehle von der Zentraleinheit zu anderen Bausteinen des Computersystems geleitet werden. Diese Steuerleitungen sind in Abb. 19 nicht angegeben. Wenn es auch die Bezeichnung vermuten läßt, so ist die Zentraleinheit nicht der Hauptbestandteil eines Computers. Schon räumlich ist der Speicher wesentlich umfangreicher. Denn nicht nur sämtliche Arbeitsgänge der Zentraleinheit beziehen sich auf den Speicher, sondern auch noch alle Eingangsinformationen und Programme werden in diesem abgelegt.

Die Bauelemente eines Computersystems bilden zusammen die Hardware, wogegen Programme als Software bezeichnet werden. Nur die Zusammenarbeit von Hard- und Software führt zu einem funktionsfähigen Computersystem.

Wird der Computer für Rechen- oder Verwaltungsaufgaben eingesetzt, dann können die Daten über eine Tastatur, Lochkarten, Magnetbänder, Lochstreifen und andere Geräte eingegeben werden, wogegen die Ausgabe des Ergebnisses über einen Drucker (Printer) oder einen Monitor (Video display unit = VDU) erfolgt.

Wird der Computer in Automaten oder Regelsystemen eingesetzt, dann kommen die Angaben von Lochkarten (z.B. bei Dreh- und Bohrmaschinen) oder Sensoren (Fühler). Der Ausgang ist dann mit Steuermotoren oder Relais gekoppelt. Alle diese Bausteine sind unter dem Begriff „Peripheriegeräte" zusammengefaßt.

Diese können nicht direkt an den Computer angeschlossen werden. Die Angaben aus den Peripheriegeräten müssen erst in eine geeignete Form gebracht werden (binäre Codeworte), damit sie vom Computer akzeptiert werden können. Die hierfür erforderliche elektronische Schaltung wird mit Interface bezeichnet.

Daten können z.B. von einem Potentiometer abgenommen werden. Die Spannung ist dabei abhängig von der Lage des Schleifkontaktes. In Abb. 20 ist das Diagramm eines linearen Potentiometers gezeichnet. Es ist dies eine analoge Darstellung von Angaben, die erst über eine besondere elektronische Schaltung, den sogenannten A/D-Wandler (Analog/Digital Wandler) als Interface in ein Digitalsignal umgewandelt werden muß. Im Beispiel nach

Abb. 20 A/D-Wandler zur Gewinnung einer binären Information aus einer veränderlichen Spannung

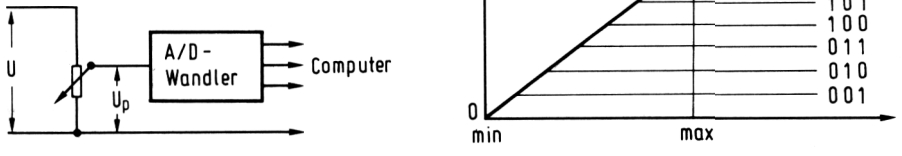


Abb. 20 erscheint am Ausgang des A/D-Wandlers in Maximalstellung des Potentiometers die Zahl 111. Je nach Schleiferstellung ergibt sich am Wandler eine entsprechende kleinere Zahl.

Es gibt auch Peripheriegeräte (wie Lochstreifenleser, Tastenfelder, Bildschirme usw.), in denen die Schrift- und Satzzeichen bereits codiert sind. Diese Codezeichen kommen dann in Form von Impulsen über eine Leitung (serielle Datenübertragung) aus dem Peripheriegerät, erfordern aber eine Anpassung an den Computer.

In den Peripheriegeräten wird meistens der ASCII-Code (*A*merican *S*tandard *C*ode für *I*nformation *I*nterchange) angewandt. Siehe auch Tabelle (Tabelle 27) im Anhang.

Mit der Kopplung von Ein- und Ausgabe an den Speicher ist sowohl die Eingabe der Daten in den Speicher als auch die Speicherung des Arbeitsergebnisses durch die Zentraleinheit in den Speicher verbunden.

Mikrocomputersysteme sind in verschiedenen Kombinationen erhältlich. Es gibt Ausführungen, die aus Computer, Tastenfeld und Bildschirm als getrennte Baueinheiten bestehen, aber auch Ausführungen mit Computer, Tastenfeld und Bildschirm als Einheit. Eine andere besteht aus Zentraleinheit, Speicher, einem Interface und evtl. einem einfachen Display und Tastenfeld auf einer einzigen Druckplatte (single board computer).

Modelle, in denen Zentraleinheit, Interfaces und Speicher auf einem Chip zusammengefaßt sind (single chip computer), werden meistens in Regelsysteme und Automaten eingesetzt.

Der englisch/amerikanische Ausdruck für eine Zentraleinheit ist „central processing unit" (CPU). Bezeichnend für den Mikrocomputer ist, daß wenigstens die CPU aus einem Chip besteht. Man spricht deshalb von einem „Mikroprozessor".

Die Möglichkeiten eines Mikrocomputers hängen im wesentlichen von der CPU-Type ab. Die Ausführungsformen sind jedoch dermaßen vielfältig, daß es nicht möglich ist, eine Beschreibung in allgemeiner Form hierüber zu geben. Es soll daher von einer Prozessor-Type ausgegangen werden, und zwar vom 6502. Später soll dann noch auf Unterschiede zu anderen Prozessoren hingewiesen werden.

9.2 Der Speicher

Ganz allgemein besteht ein Speicher aus Registern. Der maximale Speicherumfang, d.h. die Registergröße und die Registeranzahl, ist wesentlich von dem zur Anwendung kommenden Prozessor abhängig. In diesem Fall ist die Registergröße acht Bit. Ein Speicherregister kann daher ein Wort von einem Byte aufnehmen. Jedes Register ist ein Speicherplatz und jeder Speicherplatz hat seine eigene Nummer. Die Anzahl von Speicherplätzen im Processor 6502 beträgt $2^{16} = 65536$.

Die Speicherplatz-Nummer wird binär angegeben, wozu ein 16-Bit-Wort nötig ist. Das sind vier Tetraden. In hexadezimaler Wiedergabe enthält diese Zahl darum vier Ziffern (*Abb. 21*).

Die Speicherplatznummer wird „Adresse" genannt, das Ansteuern eines Speicherplatzes mit „Adressieren". Das Adressieren erfordert ein Bündel von sechzehn Leitungen, mit „Adressenbus" bezeichnet. Die Dateneingabe über einen Bus von acht Leitungen heißt „Databus" (*Abb. 22*). Das $-Zeichen vor einer Hexadezimalzahl wird im weiteren Verlauf weggelassen, da es eine Adressenbezeichnung betrifft. Wenn nicht anders angegeben, wird die Adresse stets mit Hilfe von Hexadezimalzahlen dargestellt.

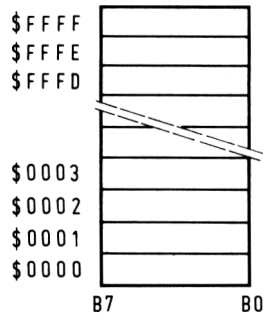


Abb. 21 Vier Hex-Ziffern ergeben eine 16-Bit-Adresse

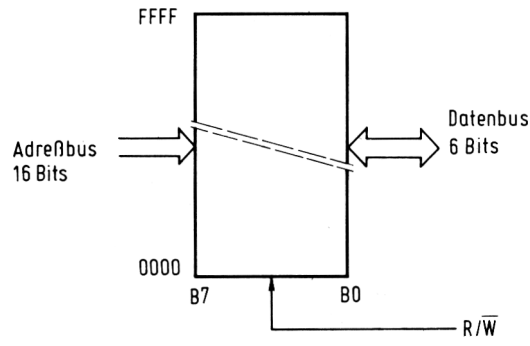


Abb. 22 Schreib/Lese-Speicher als „Black Box"

Ist ein bestimmter Speicherplatz adressiert, werden die acht Bit dieses Speicherplatzes mit dem Datenbus verbunden. Es kann nun sein, daß eine Zahl in einen Speicherplatz eingegeben oder „geschrieben" (write) werden muß – z.B. die Eingabedaten oder ein Ergebnis aus dem Prozessor – dann muß eine Zahl, die sich in diesem Speicherplatz befindet, „gelesen" werden (read), z.B. Daten nach Ausgabe. Hierzu ist eine besondere Leitung erforderlich (R/W-Leitung, Abb. 22), „niedrig" („0") bei write, und „hoch" („1") bei read. Diese Leitung ist eine der Computer-Steuerleitungen und der Prozessor kann hiermit angeben, ob der adressierte Speicherplatz „eingeschrieben" oder „ausgelesen" werden muß.

Bei einem Readbefehl wird der Inhalt des Speicherplatzes auf den Datenbus gegeben. Der Speicherplatzinhalt geht dabei nicht verloren. Bei einem Writebefehl wird die Zahl, die sich in dem Augenblick auf dem Datenbus befindet, auf einen Speicherplatz geführt, wobei der ursprüngliche Inhalt dieses Speicherplatzes verlorengeht. Adressierung und Steuerung (R/W) erfolgen ausschließlich durch den Prozessor.

Es ist nicht unbedingt gesagt, daß bei jedem Mikrocomputer mit sechzehn Adressenleitungen der Speicher 65536 Register umfaßt. Das ist nur der maximale Speicherumfang. Man spricht deshalb auch von einem Adreßumfang von 65536 Byte. Dieser wird in 64 Teile von 1024 Plätzen (2^{10}) aufgeteilt, die mit 1 KByte angegeben werden (k = 1000, K = 1024).

In grober Annäherung sind daher 64x1 KByte (65536 Speicherplätze) gleich 65 kByte (bisweilen auch mit 64 KByte) angegeben. Ein sechzehn mal so großer Adreßumfang heißt auch 1 MByte (16x65536 = 1048576). Jede Gruppe von 1024 Byte ist wieder in vier „pages" von 256 Adressen eingeteilt (*Abb. 23*). Im gesamten Speicherblock sind deshalb 64x4 = 256 pages unterzubringen. Die Seite 00 umfaßt die Adressen von 0000–00FF, Seite 01 die Adressen 0100–01FF bis Seite FF.

Es ist natürlich notwendig, daß in einem Mikrocomputer der Speicher möglichst wenig Platz einnimmt. Vor den heutigen integrierten Halbleiterspeichern wurden die „Speicherzellen" (Platz für ein Bit) aus magnetisierten Ringkernen gebildet. Die Richtung des magnetischen Feldes im Kern gab dann an, ob das Bit 1 oder 0 war. Obwohl die Ringkerne manchmal so klein waren, daß man ein Vergrößerungsglas nehmen mußte, um sie zu erkennen, nahmen die Speicher viel zuviel Platz ein. Gegenwärtig werden Halbleiterspeicher mit wenigstens 1024 Bit auf einem Chip eingesetzt, und die Entwicklung geht auf eine noch viel höhere Packungsdichte zu.

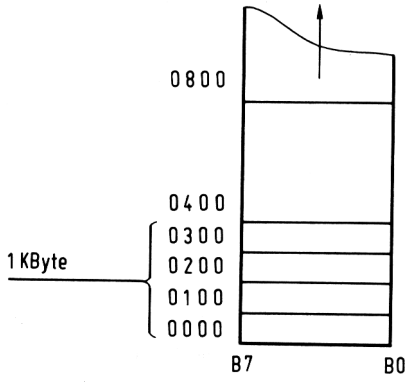


Abb. 23 Adressenbelegung eines 1-KByte-Speichers

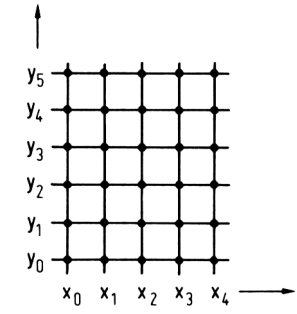


Abb. 24 Die Speicherzellen sind auf dem Chip als Matrix geschaltet

Auf dem Chip ist ebenfalls die Adreßcodierung zu finden. Die Speicherzellen sind meistens in Rasterform angeordnet (Matrix). Das ist ein Raster von sich kreuzenden x- und y-Linien (*Abb. 24*). In den Schnittpunkten befinden sich die Zellen. Eine bestimmte Zelle kann „angesprochen" werden durch die dazugehörigen x-y-Linien. Die Adreßcodierung sorgt nun dafür, daß eine bestimmte Bitkombination auf den Adreßlinien mit einer von den Zellen korrespondiert. Acht dieser Raster sind erforderlich, um einen vollständigen Speicher mit acht Bitworten zu bilden.

Die Read-Ausgänge eines Speichers sind mit Ausgangspuffern an die Datenleitungen angeschlossen. Ein Puffer ist als Verstärker anzusehen, der die notwendige Leistung liefern kann. Nun werden auf jede Datenbusleitung mehrere Puffer angeschlossen, was Schwierigkeiten ergeben kann, da die Puffer nur zwei Ausgangssituationen kennen, nämlich „High" (1) und „Low" (0). Es könnte sein, daß der eine Puffer auf High und der andere auf Low liegt, was zur Zerstörung führen könnte. Hierfür gibt es zwei Lösungen:

a) „Tristate"-Puffer. Außer den zwei Zuständen High und Low gibt es noch einen dritten, bei dem die Puffer eine unendlich große Ausgangsimpedanz haben. Dieser dritte Zustand tritt dann auf, wenn die Puffer nicht zum Auslesen von Daten aus dem Speicher benötigt werden. Ein anderer Puffer kann dann die Leitung ansteuern.

b) „Open collector"-Puffer. Dieser ist als NPN-Transistor ohne Kollektorwiderstand aufzufassen. Der Kollektor wird mit der betreffenden Datenleitung über einen Widerstand

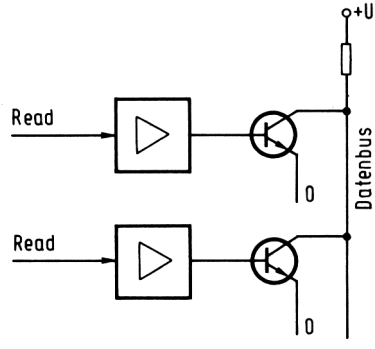


Abb. 25 Datenbus-Ansteuerung mit Open-Collector-Treibern

an +U (z.B. +5 V) verbunden. Alle Ausgangspuffer liegen an derselben Datenleitung und benutzen daher denselben Widerstand als Kollektorwiderstand (*Abb. 25*).

Leitet einer der Transistoren, dann geht die Leitung auf Low. Leitet keiner, dann sorgt der Widerstand nach +U dafür, daß die Leitung auf High-Potential liegt.

Es kann nun keine Situation auftreten, bei der einer der Transistoren zerstört wird. Wird der Datenbus nicht gebraucht, dann liegen alle Leitungen hoch.
Der Speicher eines Computers wird wie folgt unterteilt:

a) Datenspeicher
b) Stapelspeicher
c) Programmspeicher
d) Systemspeicher.

a) Der Datenspeicher wird für die Speicherung der Ein- und Ausgabe-Daten benötigt. Bei dem Prozessor 6502 ist dies besonders die „Seite" (Page) 0.

b) Der Stack ist der Kellerspeicher eines Mikrocomputers. Hier werden über den Prozessor Daten gelagert, die während des Abarbeitens eines Programmes vorübergehend aufbewahrt werden müssen. Dieser Speicher liegt beim 6502 auf Page $ 01.

c) Im Programmspeicher wird das Programm codiert gespeichert. Die Reihenfolge der Befehle in diesem Speicher ist die gleiche wie im Programm. Die Adresse der ersten Instruktion ist die Programmstartadresse.

Der vollständige Befehlssatz besteht aus der Bearbeitungs- oder Verfahrensweise, wie der Prozessor vorgehen muß, dem sogenannten Operationscode, gefolgt von der Zahl, auf die die Operation Bezug nimmt, dem „Operand" bzw. die Adresse, wo der Operand zu finden ist, meist eine Adresse aus dem Datenspeicher. Die Operation kann nur codiert in den Speicher eingegeben werden. Diese Zahl ist der Operationscode. Auf den Operationscode folgt der Operand, genauer die Adresse des Operanden.

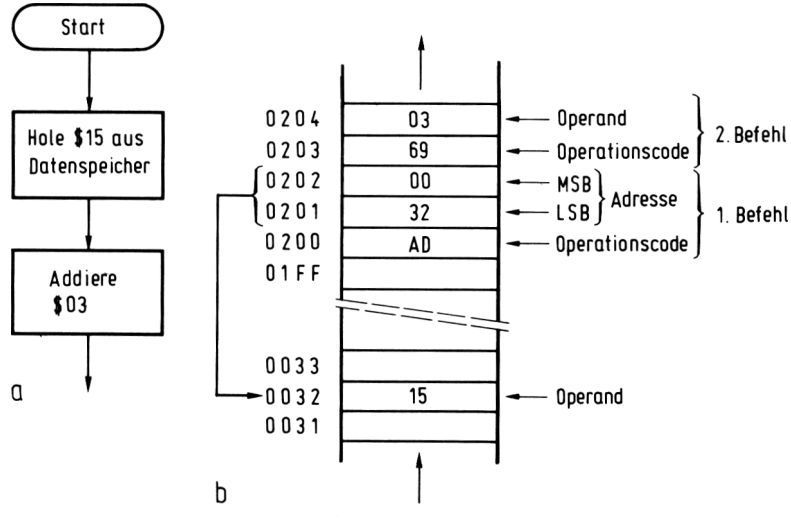


Abb. 26 Lage von Befehlen und Operanden im Speicher

In *Abb. 26a* ist das Flußdiagramm für ein Teilprogramm wiedergegeben. Dieses addiert die Zahlen $ 15 und $ 03. Die Startadresse ist $ 0200. Nach der für den Prozessor 6502 angegebenen Tabelle ist der Operationscode für die Operation „hole" $ AD. Nun muß die Adresse folgen, da die Daten im Datenspeicher stehen. Diese ist 0032 und besteht aus zwei Byte: einem Byte mit dem höchsten Stellenwert 00 (MSB = Most Significant Byte) und dem Byte mit dem niedrigsten Stellenwert (LSB, Least Significant Byte), 32. Ein Speicherplatz kann nur ein Byte enthalten, so daß für diesen Befehl insgesamt drei Speicherplätze erforderlich sind.

Der Operationscode für „Addieren" ist $ 69. Direkt anschließend folgt der Operand. In *Abb. 26b* ist die Situation für diese zwei Befehle im Speicher gezeichnet. Achte hierbei auf die Reihenfolge der Adreß-Byte des ersten Operanden, zuerst LSB und dann MSB. Der Computer arbeitet dieses Unterprogramm ab, indem er erst über die Adresse 0200 den Operationscode liest. Dann folgt der Befehl zur Addition dieses Operanden durch $ 03.

d) Der Systemspeicher enthält das Systemprogramm. Ohne dieses Programm kann der Computer nicht arbeiten. Das Systemprogramm ermöglicht das Eingeben der Daten und des Programms in den Speicher, sorgt für eine exakte Funktion des Display, d.h. steuert die Vorgänge zwischen Computer und Anwender und ist für den Komfort eines Computers maßgebend. Es wird vom Hersteller in den Computer eingegeben.

Unter den integrierten Halbleiterspeichern gibt es einige, die bei Spannungsausfall ihren Inhalt verlieren. Einen Speicher dieser Gruppe nennt man ein Random-Access-Memory (RAM, wahlfreier Speicherzugriff), unterteilt in statische und dynamische RAM.

Das statische RAM behält seinen Inhalt so lange die Versorgungsspannung eingeschaltet ist. Das dynamische RAM verliert seinen Inhalt nach einer gewissen Zeit. Als Speicherzelle dient ein Kondensator, der je nach Bitzustand geladen oder nicht geladen sein kann. Es ist eindeutig, daß die Ladung durch Leckverlust nach einiger Zeit abfließt. Diese Speicher müssen deshalb regelmäßig „aufgefrischt" werden, um die betreffenden Kondensatoren wieder aufzuladen. Der Inhalt bleibt jedoch erhalten, wenn die Speicherplätze dauernd aus- und eingelesen werden. Der Speicher, dessen Information auch nach dem Ausschalten der Speisespannung nicht verlorengeht, ist das ROM (Read Only Memory). Dieser Speicher wird durch den Hersteller eingeschrieben und kann später nicht mehr verändert werden. Es wird darauf hingewiesen, daß das Systemprogramm eines Computers in einem ROM festgelegt ist. Beim Einschalten der Speisespannung muß dieses Programm im Computer vorhanden sein, sonst besteht keine Möglichkeit, das Gerät zu bedienen.

Diesen Nachteil, daß das ROM nicht durch den Anwender geladen werden kann, hat das PROM nicht (Programmable ROM). Der Anwender kann mit einem besonderen Gerät diesen Speicher selbst programmieren. Danach verhält es sich wie ein ROM und kann nicht mehr verändert werden. Ist das Programm in ein PROM verkehrt eingeschrieben oder wird es nicht mehr benötigt, dann ist es wertlos. Das wird beim EPROM (Erasable PROM) vermieden. Dieser Speicher läßt sich durch ultraviolette Bestrahlung löschen. Das Gehäuse des IC ist mit einem Quarzglasfenster versehen, wodurch eine Bestrahlung möglich ist (gewöhnliches Glas ist undurchlässig für UV-Licht).

Ein elektrisch löschbares ROM ist das EAROM (Electrically Alterable ROM, elektrisch veränderliches ROM). Hierbei können einzelne Bits durch äußere elektrische Spannungen geändert werden. Die Anzahl der Umprogrammierungen eines EAROM ist jedoch begrenzt.

9.3 Die CPU 6502

Für die Ausführung von Programmbefehlen sind u.a. in der CPU eine Anzahl von Registern vorrätig (*Abb. 27*). Das Register„Akku" wird zur Ausführung arithmetischer Operationen benötigt und ist ein Acht-Bit-Register. Auch die x- und y-Indexregister enthalten acht Bit. Beide können als Zähler eingesetzt werden, lassen sich aber auch zum Adressieren gebrauchen.

Ein weiteres 8-Bit-Register ist der „Stackpointer" S. Am Programmanfang wird er z.B. auf $FF gesetzt. Er „zeigt" auf einen Platz in Page $ 01. Der Stackpointer adressiert also nun den Speicherplatz 01FF (*Abb. 28*). Wird durch den Prozessor dieser Speicherplatz aufgefüllt, dann wird gleichzeitig S um 1 vermindert. Der Inhalt von S wird dann 01FE. Durch das Auffüllen dieses Speicherplatzes durch den Prozessor wird der Inhalt von S 01FD usw.

Das Laden des Stack durch den Prozessor erfolgt immer von oben nach unten, dadurch ist die Adresse, die durch den Pointer angegeben wird, immer ein „unbelegter" Speicherplatz. Das Programmieren des Stack verläuft deshalb nicht willkürlich, sondern von Speicherplatz zu Speicherplatz (daher Stapelregister).

Auch das Lesen geht von Speicherplatz zu Speicherplatz vor sich und beginnt bei der zuletzt eingeschriebenen Adresse. Unter der Voraussetzung, daß die letzte eingeschriebene Adresse 01FB ist, steht der Stackpointer auf 01FA. Dies ist also ein unbelegter Speicherplatz. Beim Lesen wird erst der Stackpointer auf 1 erhöht. Das weist auf den zu lesenden Speicherplatz hin (01FB). Nun wird dieser Speicherplatz ausgelesen. Bevor die nächste Adresse eingeschrieben wird, ist der Stackpointer wieder um 1 anzuheben, worauf dann die angezielte Adresse gelesen wird. Die ausgelesenen Speicherplätze können als „unbelegt" angesehen werden (nur bei den Stack-Speicherplätzen, nicht beim Programmspeicher und meistens auch nicht bei den Datenspeichern), so daß dann der Stackpointer wieder einen unbelegten Speicherplatz anweist, den ersten im Stapel.

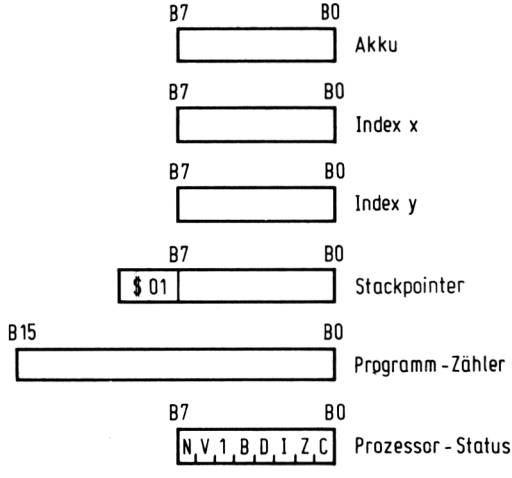


Abb. 27 Interne Register der CPU 6502

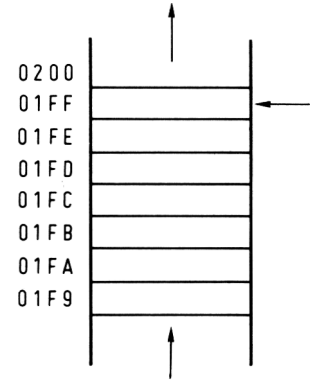


Abb. 28 Der „Stack" liegt beim 6502 im Bereich 0100 ... 01FF

Der Stack ist ein „last in, first out"-Speicher. Die zuletzt eingeschriebene Zahl wird zuerst wieder ausgelesen. Der Stackpointer gibt also immer eine Adresse auf page $ 01 an, nur das niedrigste Byte kann geändert werden.

Der „Programmzähler" (PC) oder „Befehlszähler" wird vor dem Start durch ein Programm mit der Startadresse geladen. Dieses Register muß jede Adresse ansprechen können, und da die Adresse aus sechzehn Bit besteht, muß auch der PC eine Wortlänge von sechzehn Bit verarbeiten. Der Zähler muß während des Abarbeitens des Programms durch die CPU laufend erhöht werden und gibt dann immer den Platz des Programmspeichers an, der durch die CPU gelesen werden soll. Beim Ablauf des Programms werden somit die Speicherplätze Stück für Stück nacheinander gelesen.

Der Zähler kann während eines Programms auch mit einer anderen Adresse geladen werden als nach der numerischen Reihenfolge zu erwarten wäre. Das kommt bei Sprungbefehlen vor, bei denen ein Unterprogramm (z.B. eine Subroutine) durchlaufen werden muß, das in einem anderen Speicherabschnitt untergebracht ist. Auch kann der Zähler einen Rücksprung ausführen, so daß ein gerade abgearbeitetes Unterprogramm wieder durchlaufen wird. Das kann z.B. bei einem bedingten Sprungbefehl der Fall sein (Abb. 7).

Das letzte Register ist das „Prozessor-Statusregister"(P). Dieses Register ist auch unter der Bezeichnung „Statuscoderegister" zu finden. In diesem Register sind folgende Zeichenbits zusammengefaßt: Negationsbit N, Überlaufbit V, Breakbit B, Dezimalbit D, Interrupt-Bit I, Nullbit Z und Übertragsbit C. Ein Bit in diesem Register wird nicht gebraucht, sein Wert ist immer „1". Da diese Bits den Ergebniszustand einer Ausarbeitung kennzeichnen (Ergebnis 0, Z = 1, Ergebnis negativ, N = 1 usw.) werden sie auch als „flags" bezeichnet (Z-flag, N-flag usw.).

Im Prozessor 6502 fehlt der Half carry H mancher anderen CPU-Typen. Dieses Bit übermittelt den Übertrag der ersten Tetrade eines Wortes nach der folgenden, z.B. bei einer Addition (siehe auch Abschnitt 5: Rechnen mit BCD-Code. Dort finden wir „D" für „Dezimal mode"). Wenn dort eine „1" angegeben ist, muß der Prozessor während des Rechenvorganges automatisch die für den BCD-Code erforderlichen Korrekturen ausführen, bei „0" muß gewöhnlich binär gerechnet werden.

Die meiste Betriebszeit eines Computers wird mit Warten „verschwendet". Das ist z.B. bei dem Programm nach Abb. 16 der Fall. Hier werden Weichen- und Brückenstand einer Modellbahnanlage kontrolliert. Obwohl das natürlich ständig durchgeführt werden muß, könnte der Computer sicherlich für Sekundenbruchteile andere Aufgaben erfüllen. Diese genügen dem Computer zum Durchlauf eines weiteren Programmes. Dieses „Interrupt"-Programm wird durch einen kurzen Spannungsimpuls gestartet. Erhält der Computer ein entsprechendes Signal, verläßt dieser das Programm, mit dem er beschäftigt ist, (ein Hauptprogramm) und läßt das Interruptprogramm durchlaufen. Nach dessen Verarbeitung geht der Computer wieder auf sein Hauptprogramm zurück.

Es können im Hauptprogramm jedoch Teile sein, die nicht durch einen Interrupt unterbrochen werden dürfen. In diesem Fall wird in das Bit „I" des Statusregisters eine „1" gelegt. Das Interruptsignal wird ignoriert und der Computer arbeitet weiter am Hauptprogramm. Ist I jedoch 0, spricht der Computer auf einen Interrupt an.

Die Stelle B im Statusregister wird bei einem „Break"-Befehl mit einer „1" belegt. Hierauf werden wir noch zu sprechen kommen.

In *Abb. 29* ist die interne Organisation eines Prozessors dargestellt. Außer den in Abb. 28 angegebenen Registern finden wir hier noch die Arithmetik- und Logikeinheit (ALU), das Befehlsregister, Decodierungs-, Timing- und Kontrollogik, sowie Daten- und

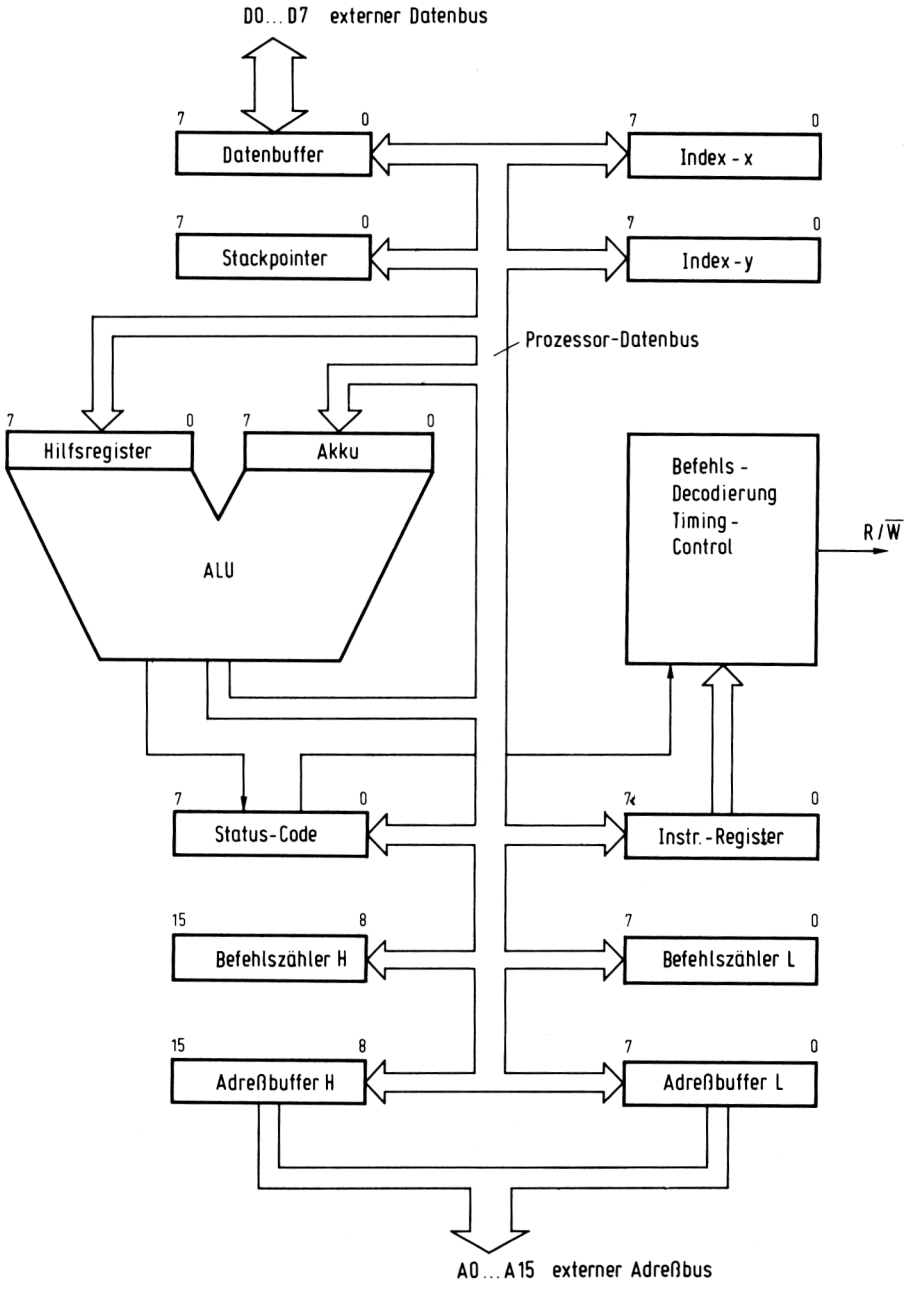


Abb. 29 Interne Struktur der CPU 6502

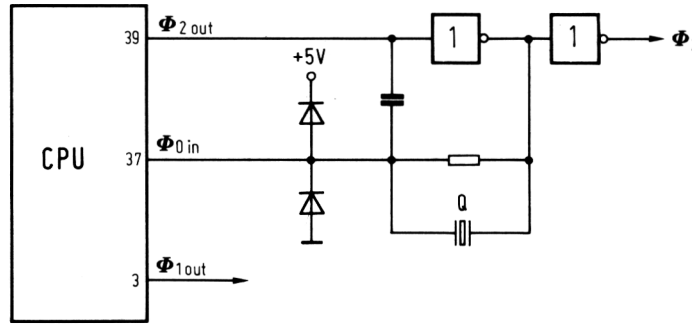


Abb. 30 Erzeugung des 1-MHz-Taktsignals

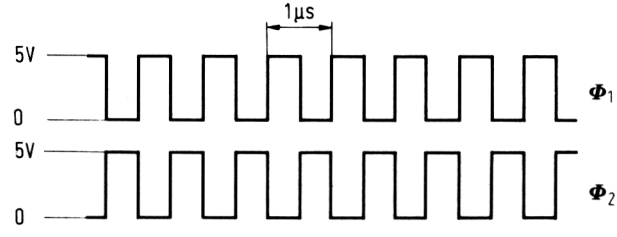


Abb. 31 Die CPU 6502 arbeitet intern mit einem 2-Phasen-Takt

Adreßpuffer. Die Datenpuffer speisen in den externen Datenbus ein (Systembus) und bilden gleichzeitig die Eingabetore für den Prozessor. Die Adressenpuffer sind mit dem externen Adreßbus verbunden. Sämtliche Puffer sind in Tristate-Ausführung.

Die für Register und ALU erforderlichen Steuersignale werden von der Decodierungs-, Timing- und Kontrolleinheit geliefert. Diese sind für das einwandfreie arithmetische und logische Arbeiten der ALU sowie für das Laden der richtigen Register mit den Daten aus dem internen Prozessordatenbus verantwortlich.

Das Abarbeiten der Prozessorvorgänge verläuft mit der Regelmäßigkeit einer „Uhr". Dem Prozessor wird dazu ein impulsförmiges Signal (Taktsignal) zugeleitet.

Der Taktgeber befindet sich beim 6502 auf dem Prozessorchip und besteht aus frequenzabhängigen Bauelementen wie R-C-Netzwerk oder Quarz (*Abb. 30*).

Das Taktsignal ist rechteckig mit einer Frequenz von z.B. 1 MHz und den Phasen Φ_1 und Φ_2 (*Abb. 31*). Die Arbeitsvorgänge im Prozessor laufen demzufolge mit einer Genauigkeit von 1 µs ab.

Der Prozessor 6502 führt gleichzeitig zwei Arbeitsvorgänge aus. In dem einen Vorgang, dem sog. externen Arbeitsablauf, wird während des Impulses Φ_1 die Adresse auf den Adressenbus gelegt. Während des darauffolgenden Impulses Φ_2 gelangen die Daten vom adressierten Speicherplatz auf den Datenbus (R / $\overline{W}$ = „1"), die vom Prozessor evtl. angenommen werden können. Steht ein Operationscode auf dem Datenbus, d.h. ist der Inhalt eines adressierten Speicherplatzes ein Operationscode und wird dieser akzeptiert, dann spricht man von einem Opcode-Fetch. Der Operationscode wird dann dem Befehlsregister zur Decodierung zugeführt. Dieser Vorgang läuft ebenfalls in einem Taktzyklus ab, somit in 1 µs.

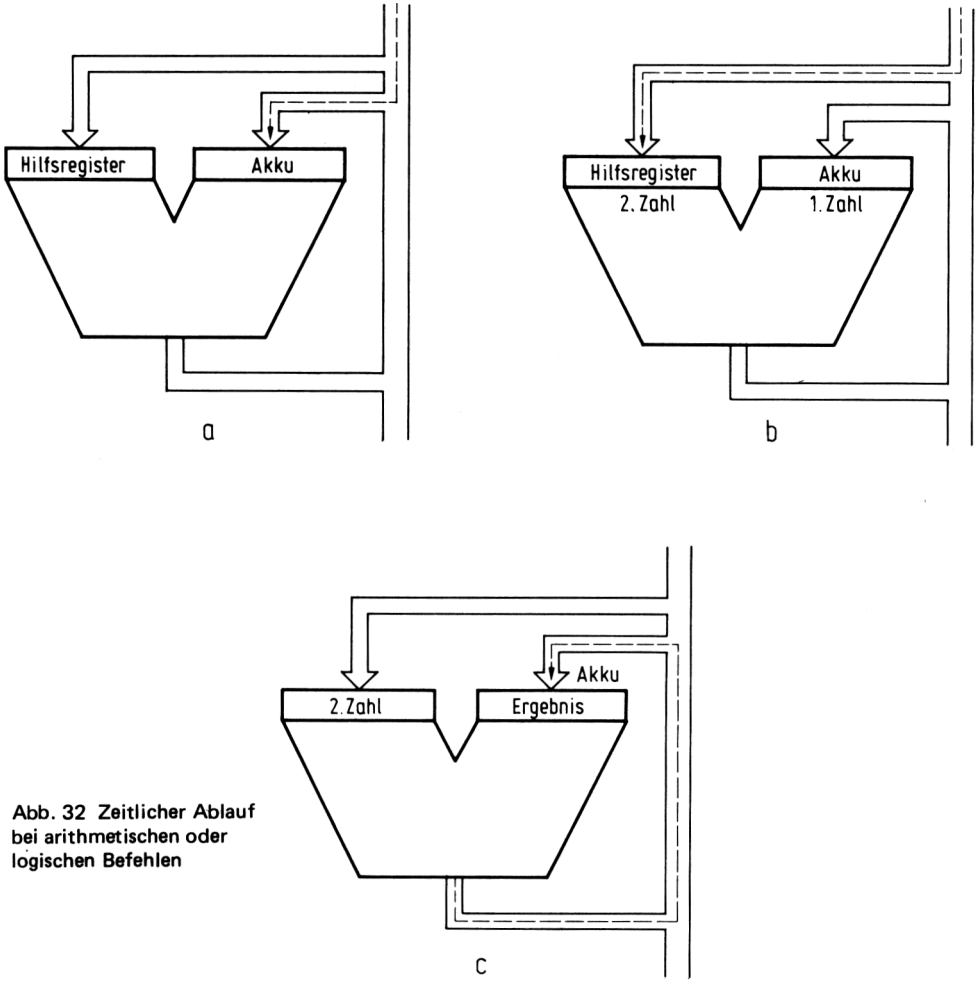


Abb. 32 Zeitlicher Ablauf bei arithmetischen oder logischen Befehlen

Gleichzeitig läuft ein interner Vorgang ab. Dieser hängt von Daten ab, die in einem vorhergehenden Zyklus vom Prozessor aufgenommen wurden, z.B. eine arithmetische oder logische Tätigkeit der ALU oder die Decodierung eines Operationscode. Während dieser Zykluszeit kann evtl. intern der Befehlszähler erhöht werden.

Das gleichzeitige Abarbeiten mehrerer Vorgänge hat den Vorteil, daß die Befehle in kürzester Zeit ausgeführt werden können.

Die ALU übernimmt arithmetische und logische Ausarbeitungen, und zwar in zwei Registern, dem Akku und einem Hilfsregister. Zuerst wird die erste Zahl im Akku eingebracht (*Abb. 32a*). Zu diesem Zweck muß der Programmierer als erstes den Befehl „Load Accu" in Form eines Opcode (Operationscode) in den Speicher eingeben. Nach dem Opcode-Fetch und der Decodierung durch den Prozessor werden die Daten von den Datentoren aus über den Prozessordatenbus in den Akku geladen.

Das Laden des Hilfsregisters erfolgt nach dem Opcode-Fetch entsprechend des auszuführenden Arbeitsvorganges (*Abb. 32b*). Den Arbeitsablauf übernimmt die ALU, das Resultat wird in den Akku geladen. Der ursprüngliche Akku-Inhalt (1. Zahl) geht dabei verloren (*Abb. 32c*).

Schließlich ist wieder ein Befehl erforderlich, um den Akku-Inhalt zurück in den Speicher zu führen (Store Accu).

Im ganzen sind also für die Abwicklung eines Arbeitsvorganges drei Befehle erforderlich:

1. Befehl: Opcode „Load Accu" + Adresse vom Operanden
2. Befehl: Opcode abarbeiten + Adresse vom 2. Operanden
3. Befehl: Opcode „Store Accu" + Adresse vom Speicherplatz.

Die Belegung der Anschlüsse der CPU 6502 geht aus *Abb. 33* hervor. Die Adressenleitungen haben die Anschlüsse 9 bis 20 und 22 bis 25, die Datenleitungen mit den Anschlüssen 26 bis 33.

Vss ist mit 1 und 21, Vcc (+5 V ±5%) mit 8 verbunden. Die Anschlüsse 3, 37 und 39 sind für das Taktsignal vorgesehen (siehe auch Abb. 30). Wenn EPROM oder EAROM benötigt werden, liegt der Eingang RDY (ready) auf Anschluß 2. Die genannten Speicher haben eine relativ lange Zugriffszeit und können in Systemen, die mit höchster Geschwindigkeit arbeiten, u.U. nicht eingesetzt werden. Indem man den RDY-Anschluß während

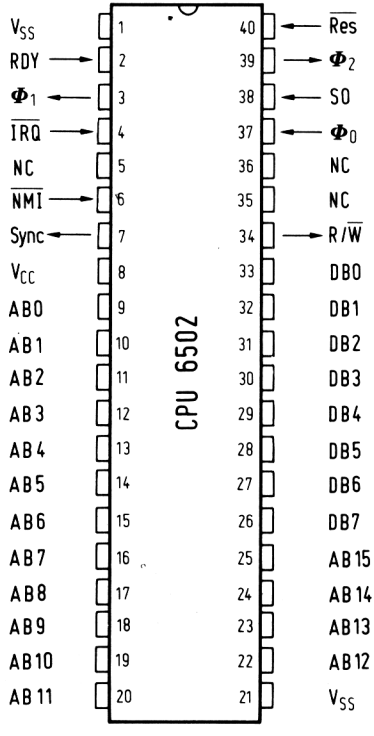


Abb. 33 Der 6502 ist in einem 40-Pin-Gehäuse untergebracht

der Impulsdauer Φ_1 und $R/\overline{W} = 1$ auf Low-Pegel legt, wird der Fetch-Zyklus des Prozessors verzögert, bis die Daten aus dem Speicher auf den Datenleitungen verfügbar sind.

Mit dem Eingang IRQ (interrupt request) auf Anschluß 4 kann ein laufendes Programm für einen Interrupt unterbrochen werden. Ein Interrupt wird dadurch hervorgerufen, daß dieser Anschluß auf Low gelegt wird, während I im Statusregister „0" ist. Dieses Interrupt-Signal ist „maskierbar" und wird negiert, wenn I = „1" ist.

Auch auf Anschluß 6 kann ein Interrupt-Signal gegeben werden. Wiederum ist der Anschluß auf Low-Pegel zu legen. Dieser Interrupt ist jedoch nicht maskierbar; er wird, unabhängig vom Inhalt von I im Statusregister, ausgeführt (NMI, Non-Maskable Interrupt).

Auf Anschluß 40 finden wir den Reset-Eingang (RES). Dieser dient u.a. zum Zurücksetzen der Register in die CPU. Beim Einschalten ist der Inhalt dieser Register beliebig. Der Prozessor muß in einem Wartezyklus ein Programm durchlaufen lassen (Systemprogramm) und auf Daten zum Laden des Speichers mit einem Programm warten. Ein Wartezyklus ist z.B. „loop" im Programm nach Abb. 16.

Indem man RES kurz auf Low legt, wird nach sechs Taktzyklen der Adressenbus auf FFFC gesetzt. Auf dieser Adresse steht das niedrigste Byte der Startadresse des Systemprogramms. Dieses Byte wird im Befehlszähler auf Low-Pegel gebracht; die Datenleitungen gehen auf FFFD. Auf dieser Adresse steht das höchste Byte der Startadresse, die im Befehlszähler auf „H" Pegel stand. Der Inhalt des Befehlszählers erscheint nun auf den Adreßleitungen und der Prozessor arbeitet das Programm mit dieser Adresse beginnend ab. Sowohl die Startadresse in FFFC und FFFD als auch das Systemprogramm müssen in einem ROM vorhanden sein.

Der Prozessor wird mit einem Low-Impuls am Anschluß RES gestartet und findet auf zwei Speicherplätzen eine Adresse, wo der erste Befehl des Programms steht.
Der Inhalt dieser Speicherplätze wird mit „Vektor" bezeichnet. Die 6502 enthält drei Vektoren, den

NMI-Vektor auf FFFA und FFFB, den
Reset-Vektor auf FFFC und FFFD, und den
IRQ-Vektor auf FFFE und FFFF.

Bei einem NMI-Interrupt wird der laufende Befehl noch in der CPU ausgeführt, Befehlszähler und Statusregister auf den Stack gerettet und der Inhalt der Speicherplätze FFFA und FFFB in den Prozessor geladen. Diese Daten bilden die Adresse des Interrupt-Programms in der Reihenfolge niedrig/hoch (LSB/MSB). Das Interrupt-Programm wird nun durchlaufen. Am Ende finden wir die Aufforderung „zurück vom Interrupt" (RTI). Der ursprüngliche Inhalt von Befehlszähler und Statusregister wird vom Speicher geholt, so daß der Prozessor wieder auf die Adresse gehen kann, wo das Hauptprogramm verlassen wurde. Dadurch, daß der NMI-Anschluß „flankengetriggert" ist, erfolgt nur einmal ein Interrupt, auch wenn dieser Anschluß weiter auf „L" gehalten wird. Bevor auf das Interrupt-Programm gesprungen wird (Startadresse in FFFE und FFFF), setzt die CPU automatisch das Interruptflag I. Durch Befehl „zurück vom Interrupt" wird I wieder O, so daß es jetzt Zeit ist, IRQ auf H zu legen, sonst erfolgt wieder ein Interrupt.

Das Synchronisierungssignal auf Anschluß 7 wird benötigt, um den Fetch-Zyklus des Prozessors zu kennzeichnen. Während des gesamten Zyklus des Opcode-Fetch liegt dieser Anschluß auf „H". Über Anschluß 38 (SO, *s*et *o*verflow) kann das V-Bit im Statusregister beeinflußt werden.

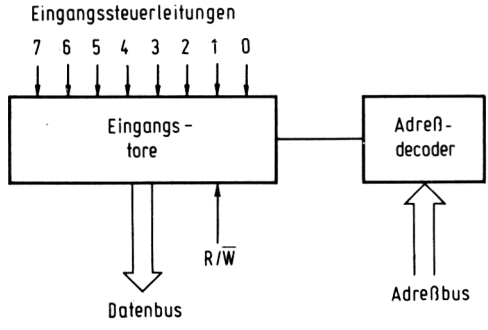


Abb. 34 Aufbau eines Eingangsports

9.4 Ein/Ausgabe

Mit dem Beispiel nach Abb. 16 lassen sich die Bedingungen für die Ein/Ausgabe erkennen, die diese als Schalter zwischen Datenbus und Ein/Ausgabe-Daten erfüllen muß. Die Eingabedaten sind über den Datenbus nach der CPU zu leiten, und zwar dadurch, daß die Eingangsschaltung (Eingabekanal) als Register durch den Datenbus ausgelesen werden kann. Diesem Register muß ebenfalls eine Adresse zugewiesen werden, so daß dieses – vom Prozessor aus gesehen – als Speicherplatz angesehen werden kann.

Der Inhalt dieses Registers wird jedoch direkt bestimmt durch den Zustand der Eingangssteuerleitungen, die mit diesem verbunden sind.

In *Abb. 34* ist das Prinzip dargestellt. Ein Adreßdecoder gibt ein Signal an das Eingangsregister, sofern die richtige Adresse auf dem Adreßbus liegt. Der Inhalt der Eingangskanäle wird dann bei $R/\overline{W}$ = „1" durch den Datenbus ausgelesen.

Das Eingangsregister enthält acht Tore, (für jedes Bit ein Tor). Es können deshalb ebenso viele Eingänge angeschlossen werden. Diese Eingangssteuerleitungen haben nur zwei Zustände H und L (1 und 0). Die hier angeschlossenen Eingangstore nehmen immer den Wert der betreffenden Steuerleitung an. Durch Ändern des Wertes der Steuerleitung ändert sich auch der Wert des Eingangstors.

Im Prinzip wird durch den Datenbus der Zustand der Eingangssteuerleitungen gemessen. Durch Maskieren kann der Zustand einer bestimmten Steuerleitung festgelegt werden. Es soll z.B. der Wert der dritten Steuerleitung bestimmt werden. Die Leitungen stellen zusammen ein Binärwort von acht Bit dar, z.B. 00110100 (die Steuerleitungen 2, 4 und 5 liegen auf Pegel „H", die anderen auf „L").

Jetzt erfolgt Maskierung mit 00000100:

Eingangskanal I	00110100
Maske M	00000100
I∧M	00000100 (Z = 0)

Am Wert von Z im Statusregister ist zu erkennen, daß das Ergebnis nicht 0 ist (Z = 0). Die dritte Leitung (2^2) liegt deshalb auf „H".

Die Eingangstore haben stets den gleichen Wert wie die Eingangssteuerleitungen. An das Ausgangsregister wird eine andere Forderung gestellt: Ist ein bestimmter Platz im Ausgangsregister einmal durch den Datenbus eingelesen, kann sein Zustand nur dann

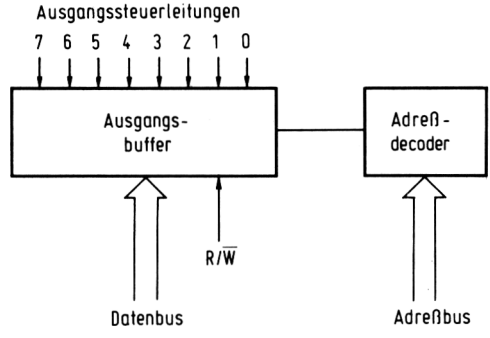
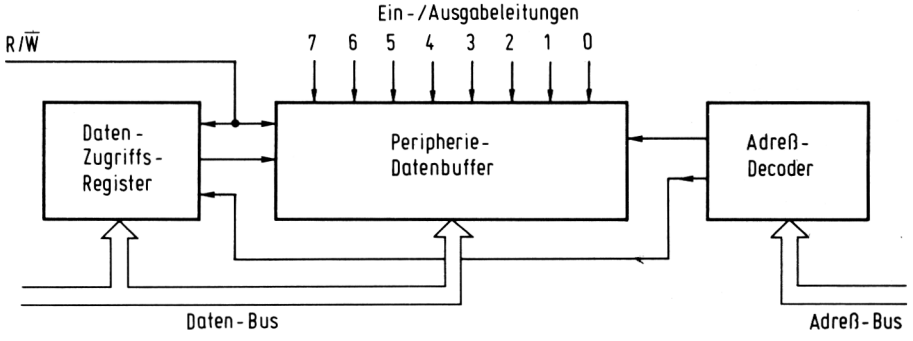


Abb. 35 Aufbau eines Ausgangsports

Abb. 36 Hier kann der Programmierer bestimmen, welche Leitungen als Ein- und Ausgänge dienen sollen



geändert werden, wenn ein Bit mit invertiertem Wert eingelesen wird. Das heißt, eine „1" in einer Registerstelle bleibt so lange erhalten, bis eine „0" eingelesen wird. Dann wird dieser Wert festgehalten.

Jedes Register ist ein Puffer und besteht im wesentlichen aus einem „Flipflop" mit zwei Transistoren, von denen der eine leitet, der andere sperrt. Die Transistoren halten diesen Zustand gegenseitig ein. Durch einen Schreibbefehl kann dieser Zustand evtl. geändert werden. Der leitende Transistor sperrt, der andere leitet. Auch in dieser Situation bleiben die Transistoren stabil. Einer der beiden Transistoren bestimmt den Zustand der Steuerleitung.

Die Notwendigkeit des stabilen Zustandes demonstriert das Programm in Abb. 16. Da hier von „Loop" die Rede ist, wird nur das Register S mit einer „1" oder einer „0" eingeschrieben (Schreiben nach einem Ausgangstor). Das grüne oder rote Signal muß ständig leuchten und das Ausgangstor konstant auf „1" oder „0" liegen.

Die Organisation der Puffer gleicht der der Eingangstore. Auch dieses Register erhält eine Adresse zugewiesen (eine andere als das Eingangsregister) und kann durch den Datenbus eingelesen werden, sofern die richtige Adresse auf dem Adreßbus steht. Der Adreßdecoder gibt dann ein Signal an die Ausgangspuffer und wenn $R/\overline{W}$ = „0" ist, wird das Wort, das zu dieser Zeit im Datenbus steht, in das Ausgangsregister eingeschrieben (*Abb. 35*).

Soll die Belegung der verfügbaren Steuerleitungen, ob Ein- oder Ausgang, selbst bestimmt werden, dann können die Schaltungen nach Abb. 34 und 35 kombiniert werden (*Abb. 36*).

Der Peripherie-Datenpuffer hat die Funktion sowohl eines Eingangs- als auch Ausgangsregisters. An dieses Register sind acht Steuerleitungen angeschlossen, und es erhält eine bestimmte Adresse zugewiesen, z.B. $1700.

Jede dieser acht Steuerleitungen kann sowohl als Eingangs- als auch als Ausgangssteuerleitung geschaltet werden. Hierzu ist ein Datenrichtungsregister erforderlich. Auch dieses Register erhält eine Adresse zugewiesen (z.B. 1701).

Wenn ein bestimmter Platz in diesem Register eine „1" oder „0" erhält, wird die hiermit übereinstimmende Steuerleitung Aus- bzw. Eingang.
Wird z.B. in die Adresse 1701 (Data Direction Register) das Wort

01101000

eingeschrieben, dann werden die Steuerleitungen 3, 5 und 6 Ausgabe, die anderen Eingabe. Durch das Adressieren mit 1700 (Peripherie-Datenpuffer) wird nur nach den Ausgangssteuerleitungen geschrieben. Es wird wohl ein 8-Bit-Wort auf den Datenbus gelegt, aber nur die Bits, die den Ausgangssteuerleitungen entsprechen (hier Bits 3, 5 und 6) werden geschrieben.

Beim Lesen dieser Adresse werden nur die Eingangssteuerleitungen ausgelesen. Nur die Bits, die mit den Eingangssteuerleitungen korrespondieren, sind verwertbar, die anderen dagegen sind wertlos (don't care).
Beispiel:

1xx0x101

Die Leitungen 3, 5 und 6 sind Ausgangsleitungen, deshalb sind die Bit 3, 5 und 6 „don't care".

9.5 Interface

In Abb. 20 haben wir uns bereits mit einer bestimmten Interface-Form beschäftigt. Es handelte sich dort um ein analoges Signal aus einem Potentiometer, das für den Computer in eine binäre Zahl umgesetzt werden muß, wobei ein bestimmter binärer Wert zu-

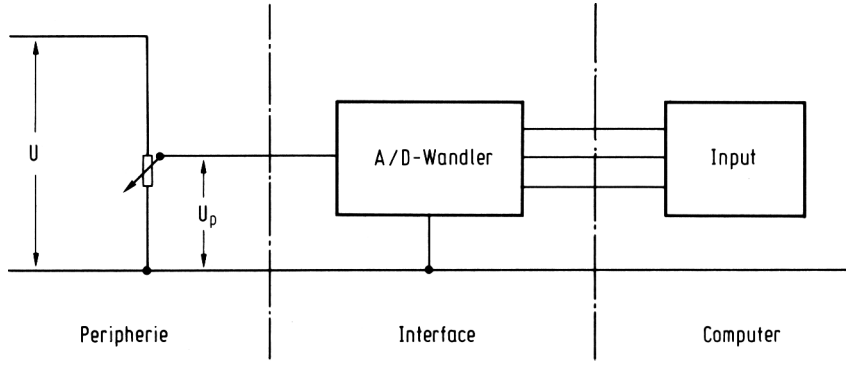


Abb. 37 Anschluß eines A/D-Wandlers an den Mikrocomputer

verlässig die Spannung u_p wiedergibt. In *Abb. 37* ist der A/D-Wandler die Schnittstelle zwischen Computereingang und Potentiometer-Ausgang. Der A/D-Wandler ist hier das Interface. Die Wortgröße aus dem Wandler bei einem Maximalwert von u_p bestimmt die Leitungszahl, die zwischen Wandler und Eingang angeordnet werden muß. Im Beispiel Abb. 20 ist das größte Wort 111. Hierfür sind drei Eingangsleitungen erforderlich.

Es handelt sich hier um die „Eingabe" von Daten. Es geht auch umgekehrt, daß Output-Daten mit einem D/A-Wandler von binär in analoge Signale umgesetzt werden müssen. Ein einfaches Interface sehen wir an der Brücke und der Weiche aus Beispiel Abb. 15. Brücke und Weiche können beide einen Schalter bedienen. In diesem Fall ist der offene Schalter „1", der geschlossene „0" (auch umgekehrt).

Der Schalter stellt nun das Peripheriegerät dar, wie in Schema *Abb. 38* gezeigt wird. Ist der Schalter geschlossen, dann liegt die Eingangsleitung an 0 V und ist Low. Ist der Schalter offen, dann wird die Steuerleitung über den Widerstand „H".

Am Schaltungsausgang muß bei S = „1" eine grüne Signallampe und bei S = „0" eine rote Signallampe brennen. Als zweckmäßig erweist sich da eine Schaltung mit zwei Transistoren (*Abb. 39*). Liegt der Ausgang hoch, leitet Transistor T1 und die grüne Signal-

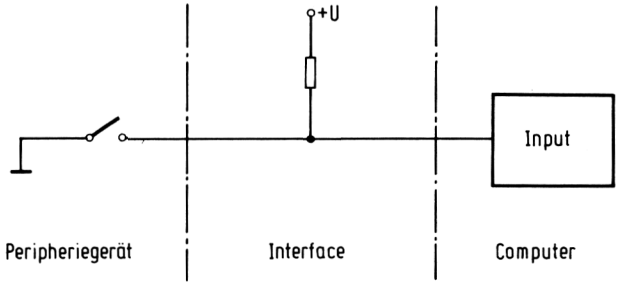


Abb. 38 Ein Pull-up-Widerstand zieht den Eingang auf High-Potential, wenn der Schalter offen ist

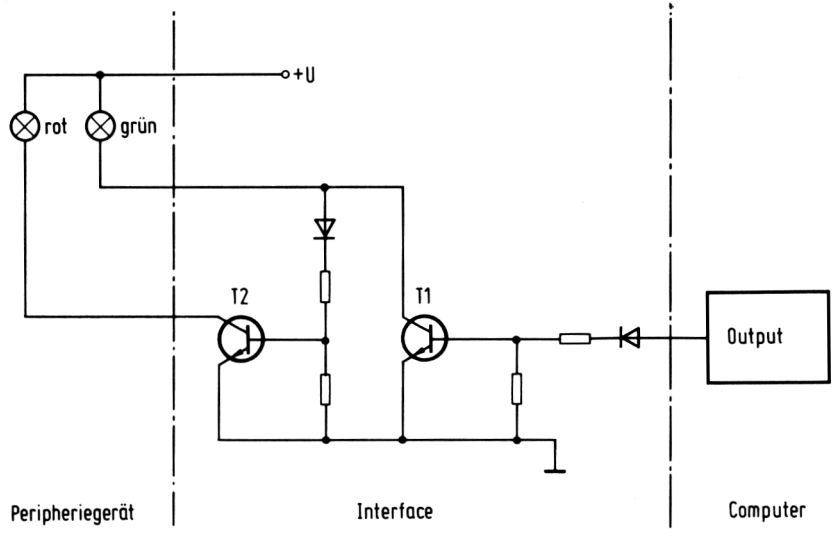


Abb. 39 Ansteuerung von zwei Signallampen mit Transistoren

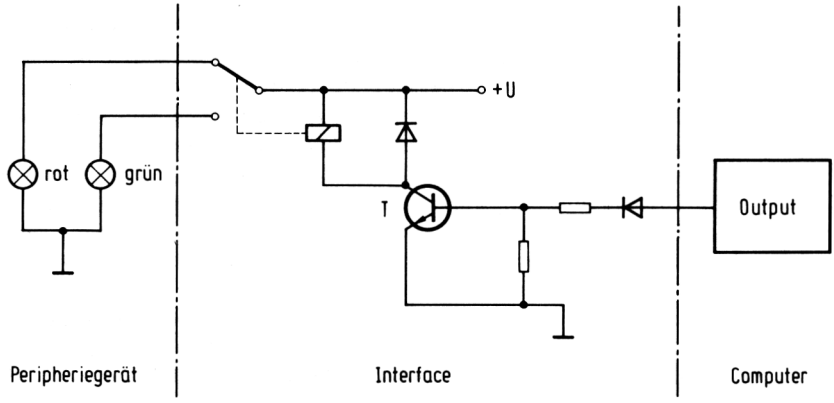


Abb. 40 Gleiche Schaltung wie in Abb. 39, jedoch mit Relais

lampe leuchtet auf. Die Kollektorspannung von T1 liegt dann auf niedrig, so daß T2 sperrt und die rote Signallampe erlischt. Wird der Ausgang niedrig, sperrt T1. Die Kollektorspannung liegt dann wieder hoch und T2 leitet. Die grüne Signallampe erlischt und die rote leuchtet auf.

Für die Signallampenschaltung läßt sich auch ein Transistor mit Relais und Wechselkontakt verwenden (*Abb. 40*). Eine andere Interface-Art ist für den Datentransfer über eine Leitung zu einem Peripheriegerät, z.B. Bildschirm, erforderlich. Das aus mehreren Bits bestehende Code-Wort (z.B. 7 bei ASCII) muß mit einer zumindest gleichen Anzahl paralleler Steuerleitungen vom Computer aus übertragen werden. Die Daten gelangen aber nur über eine Steuerleitung zum Peripheriegerät. Die Bits der Codeblöcke müssen daher nacheinander (seriell) auf diese Leitung gelegt werden.

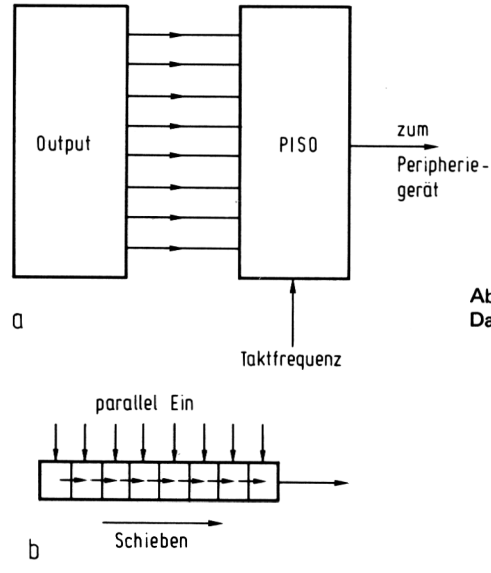


Abb. 41 Realisation einer seriellen Datenausgabe mit PISO

In *Abb. 41a* ist als Interface ein sogenannter Parallel/Serienumsetzer (PISO, parallel ein-seriell aus) eingesetzt. Hierin wird das Codewort aus dem Computer in ein Register übernommen und Bit für Bit im Rhythmus des Taktgebers auf die Ausgangssteuerleitung „geschoben" (*Abb. 41b*). Ist ein Codeblock vollständig übertragen, wird ein weiterer in das PISO-Register übernommen und der Reihe nach (seriell) abgearbeitet.

Einem Codewort mit sieben Bits, wie bei ASCII, wird noch ein besonderes Bit hinzugefügt. Dieses sog. Paritätsbit kontrolliert z.B., ob in jedem Wort eine gerade Anzahl Bit 1 ist. Liegen z.B. in einem Codewort drei Bits auf 1, dann ist das Paritätsbit ebenfalls „H". Es liegen dann vier Bits auf 1. Hat ein Codeblock sechs Bits auf 1, dann ist das Paritätsbit 0. Erhält das Peripheriegerät jedoch einen Codeblock mit einer ungeraden Anzahl 1-Bits, dann liegt bei der Übertragung ein Fehler vor.

Dieses Kontrollsystem kann aber auch von einer ungeraden Anzahl von Bits ausgehen, der Vorgang bleibt jedoch derselbe.

Beim Decodieren des Codewortes verwirft das Peripheriegerät das Paritätsbit und ermittelt ausschließlich den Wert der sieben Code-Bits.

In gleicher Weise wie mit PISO gibt es auch eine umgekehrte Arbeitsweise, und zwar seriell ein und parallel aus (SIPO). Dieses Interface ist für den Anschluß eines Peripheriegerätes mit seriellem Ausgang an einen Computereingang vorgesehen (Terminal-Tastatur).

Ein besonderes Interface in beide Richtungen für parallel/serielles Umsetzen ist der UART (*U*niversal *A*synchronous *R*eceiver *T*ransmitter). Dieser Baustein kann evtl. direkt am Datenbus angeschlossen werden.

9.6 Hintergrundspeicher

Zur Programmübertragung in den Computerspeicher muß der Speicher durch die CPU eingelesen werden können. Hierzu ist nur ein RAM geeignet. Das führt aber dazu, daß beim Einschalten des Computers das Programm jedesmal wieder neu eingeschrieben werden muß. Besonders bei langen Programmen ist dies sehr hinderlich und zeitraubend, da Befehl auf Befehl über ein Tastenfeld eingegeben werden muß.

Das Programm kann jedoch auf eine Kassette, eine Magnetplatte, Lochkarten oder Lochstreifen übertragen werden und mit einer hierfür geeigneten Apparatur in den Computer geladen werden. Da hier die Rede vom Speichern eines Programms außerhalb des Computers ist, spricht man von Hintergrundspeichern. Dabei gibt es zwei Möglichkeiten:

a) Der Hersteller liefert das betreffende Programm in einem derartigen Hintergrundspeicher an den Benutzer.

b) Der Anwender entwirft selbst das Programm, führt dieses mit Hilfe einer Tastatur in das RAM des Computers ein und transferiert es dann in einen Hintergrundspeicher, um es bei Bedarf wieder abzurufen.

Der bekannteste Hintergrundspeicher ist wohl die Kassette. Mit einem einfachen Recorder ist es möglich, Programme aufzunehmen und in den Computer einzugeben. Zur Bandaufnahme werden die Speicherplätze, in denen sich das Programm befindet, Wort für Wort gelesen und seriell ausgeleitet. Die Bits werden in ein Tonsignal umgesetzt, wobei aus der Signalfrequenz erkennbar ist, ob ein Bit „1" oder „0" ist.

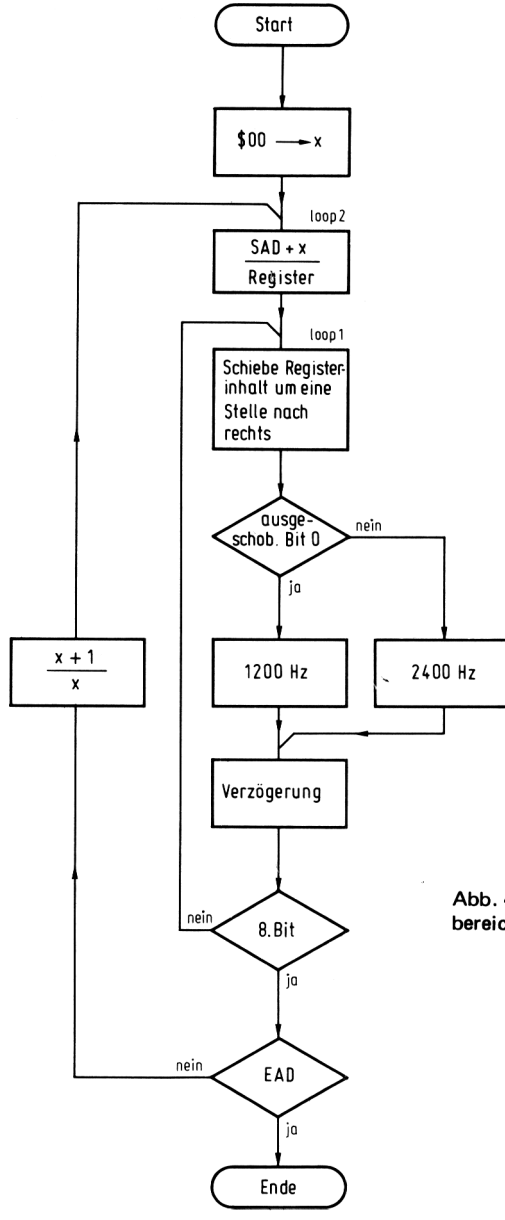


Abb. 42 Ausgabe eines Speicherbereichs auf Kassette

Ein Programm kann vereinfacht wie in *Abb. 42* dargestellt ablaufen. Zuerst wird ein Zähler auf 0 gesetzt ($ 00→ x). Danach wird der Inhalt des ersten Speicherplatzes – die Programmstartadresse ist der erste Speicherplatz – in ein Register eingegeben (SAD bezeichnet *S*tart *AD*resse, x = 0). Der Inhalt dieses Registers wird nun eine Stelle nach rechts verschoben. Das erste Bit schiebt sich dabei gleichsam aus dem Register. Der Wert

dieses Bit kann ermittelt werden und in Abhängigkeit von diesem Wert wird ein Niederfrequenzsignal erzeugt, 2400 Hz für „1" oder 1200 Hz für „0". Dieses Niederfrequenzsignal bleibt für eine gewisse Zeit durch eine Verzögerungsschaltung eingeschaltet. Innerhalb der Verzögerungszeit wird das Signal auf die Ausgangssteuerleitung zum Recorder gelegt. Das Programm springt nun auf loop 1, das Register wird wieder eine Stelle nach rechts verschoben usw. Das dauert so lange, bis sämtliche acht Bits aus dem Register verschoben sind. Nun wird geprüft, ob die zu bearbeitende Adresse die Endadresse ist (EAD ist End-Adresse). Beim ersten Mal ist das nicht der Fall; das Programm geht auf loop 2 zurück, nachdem der Zähler um 1 angehoben wurde.

Dieser Zyklus wird stets durchlaufen, und zwar bis zu einer Adresse, die um eins höher liegt als die vorhergehende, bis also die Endadresse abgearbeitet ist. Dann hält das Programm an.

Das gesamte Programm kann einschließlich der Erzeugung der 1200- und 2400-Hz-Signale vom Computer ausgeführt werden. Das mit zwei verschiedenen Frequenzen arbeitende System wird mit FSK (*F*requency *S*hift *K*eying) bezeichnet. Die Frequenzwerte sind dem Magnetbandkassetten-System angepaßt (CUTS-Codierung, CUTS: *C*omputer *U*sers' *T*ape *S*ystem, bzw. Kansas-City-Standard).

Dieses Programm ist ein Beispiel für die Möglichkeit, Binärworte mit Hilfe von Software seriell einzugeben. Die Schleife loop 1 kommt hier zur Anwendung, die die Werte Bit für Bit im Verzögerungsrhythmus ermittelt. Daß die Zeichen im FSK-System, anstelle von H und L, wiedergegeben werden, ist im Prinzip ohne Bedeutung.

Das Gegenteil hiervon ist die Parallel/Serienumsetzung nach Abb. 41a. Der Vorteil dieser Hardware-Ausführung liegt darin, daß der Computer gleichzeitig Daten in die Peripheriegeräte übertragen und ein Programm abwickeln kann. Hier besteht übrigens auch die Möglichkeit, ein FSK-System über einen PISO-Ausgang an einen Modulator anzuschließen. Dieser Modulator befördert dann die Bits mit der richtigen Frequenz zum Peripheriegerät.

Die Programmübertragung aus einer Kassette erfordert eine Demodulation, um die Bits wieder in H und L zu übersetzen. Die für die Modulation und Demodulation erforderliche Elektronik wird Modem genannt, entsprechend *MO*dulator-*DEMO*dulator.

Außer der CUTS-Codierung kommen auch noch andere Systeme zur Anwendung, z.B. ein System, bei dem die Bitwiedergabe sowohl mit 3700 Hz als auch 2400 Hz erfolgt, z.B. bei einer „0" werden 2/3 eines Zeitintervalls mit 3700 Hz und 1/3 mit 2400 Hz moduliert (z.B. KIM, *Abb. 43*).

Der Vorteil dieses FSK-Systems liegt darin, daß es fast unabhängig von der Bandgeschwindigkeit des Recorders arbeitet. Die Wiedergabe-Bandgeschwindigkeit kann also höher oder tiefer liegen als bei der Aufnahme. Bei der Demodulation wird zuerst die

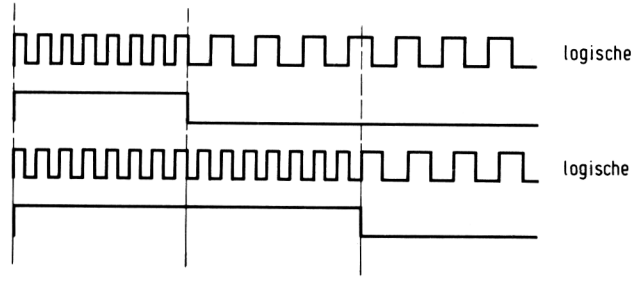


Abb. 43 Aufbau des KIM-Kassettenformats (1 Bit)

Mittenfrequenz des vom Recorder wiedergegebenen Signals festgestellt, das ist bei der CUTS-Codierung $\frac{2400 + 1200}{2}$ = 1800 Hz. Läuft der Recorder z.B. bei der Wiedergabe langsamer als bei der Aufnahme, liegt die Wiedergabefrequenz tiefer, damit also auch die Mittenfrequenz. Ist diese einmal ermittelt, können die logischen 1 und 0 jederzeit erkannt werden.

Außerdem wird das Band vor dem eigentlichen Programm mit einer Reihe von Synchronisierungszeichen versehen, mit deren Hilfe die Mittenfrequenz bestimmt werden kann. Dann folgt eine „Identification number", um bei Bändern mit mehreren Programmen das gewünschte Programm auswählen zu können, danach die Startadresse des Programms, um dieses auf den richtigen Platz im internen Speicher laden zu können. Mit „internen Speicher" ist der Speicher des Computers gemeint.

Der Inhalt der Register wird manchmal bei der Bandaufnahme über Tetraden in ASCII-Zeichen übersetzt. Zur Übertragungskontrolle wird ein Paritätsbit zugefügt.

Zur Anwendung kommen nur Kassettenbänder bester Qualität. Bei schlechter Qualität können durch sog. Drop-Outs Zeichen verlorengehen. Als besondere Kontrolle erhält das Bandende eine Prüfsumme. Der Computer vergleicht diese Zahl mit der Summe aller eingegangener Zeichen und kann dann feststellen, ob ein Programm fehlerlos übertragen wurde.

Bei einer anderen Kontrollmethode wird das Programm zweimal aufgenommen. Durch das Abspielen der zwei Versionen werden diese gegenseitig verglichen, wobei evtl. Fehler festgestellt werden können (z.B. CBM, VC-20, MZ-80).

Ein weiterer Hintergrundspeicher, der auf der magnetischen Registrierung von Informationen beruht, ist die Floppy Disk, eine runde, weiche Kunststoffscheibe mit Eisenoxidbeschichtung. Zum Schutz der Scheibe befindet sich diese in einer Art Schutzhülle, die innenseitig mit Filz verkleidet ist, um das Ansammeln von Staubteilchen auf der Scheibe zu verhindern. In der Schutzhülle ist ein Schlitz angebracht, in dem sich ein Aufnahmekopf radial bewegen kann. In ähnlicher Weise wie eine Band-Kassette enthält die Platte Spuren mit den Informationen. Diese Spuren verlaufen konzentrisch zum Mittelpunkt. Mit einer Mechanik wird der Aufnahmekopf über eine Spur zur Aufnahme oder Wiedergabe der Information geführt. Die Bitdichte ist groß bei einer relativ hohen Umdrehungsgeschwindigkeit, so daß gegenüber dem Kassettenband sehr kurze Zugriffszeiten erreicht werden. Lochkarten und Lochstreifen erfordern eine besondere Apparatur für das Programmieren, im allgemeinen wird mit ASCII-Zeichen gelocht. Ist ein Programm

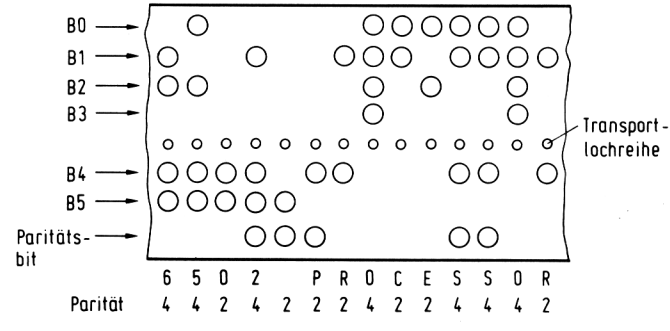


Abb. 44 Lochung eines Achtkanal-Lochstreifens

auf Lochkarten zu verbessern, dann können eine oder mehrere Karten einfach ersetzt werden. Die Änderung von Lochstreifen ist nach dem Lochen nicht mehr möglich.

Ein kleiner Lochstreifenabschnitt für 6-Bit-ASCII (reduzierter Code) ist in *Abb. 44* dargestellt. Es stehen ASCII-Versionen mit 6, 7 oder 8 Bits pro Wort zur Verfügung – je nach gewünschter Zeichenzahl (64, 128 oder 256). Die unterste Lochreihe des abgebildeten Lochstreifens ist für die Paritätsbits vorgesehen. Es liegt hier eine gerade Parität vor. Der Bandtransport erfolgt durch sog. Transportlöcher, in die ein Zahnrad eingreift. Heute wird der Lochstreifen meist optisch ausgelesen. Der Lochstreifenleser ist wesentlich einfacher als ein Lochstreifenstanzer. Zum Lochen ist eine recht komplizierte Apparatur erforderlich.

10 Befehle

10.1 Der Befehlssatz

Die Beschreibung von Befehlen ist für den allgemeinen Gebrauch sehr unhandlich. Ein Befehl ist z.B.:
Transfer index-x-register to accu.

Es sind deshalb für die Befehle Abkürzungen im Gebrauch, die so gewählt sind, daß deren Bedeutung auch wieder rückwärts gefunden werden kann:

TXA: *T*ransfer Index-*X*-Register to *A*ccu
LDA: *L*oa*D* *A*ccu.

Die leicht zu merkende Abkürzung wird mit „Mnemonic-Symbol" bezeichnet. Die gesamte Anzahl von Befehlen der CPU heißt „Befehlssatz". Jeder CPU-Typ hat leider einen anderen Befehlssatz.
Die Befehle können nach der Aufgabenart in Gruppen eingeteilt werden:

a) Übertragungsbefehle.
b) Arithmetische oder Rechenbefehle.
c) Vergleichsbefehle.
d) Logische Befehle.
e) Schiebebefehle.
f) Bedingte Sprungbefehle.
g) Unbedingte Sprungbefehle.
h) Übrige.

10.2 Übertragungsbefehle

Bei den Übertragungsbefehlen handelt es sich um den Datentransfer zwischen zwei Registern. Es gibt drei Arten von Übertragungsbefehlen:

Load-Store-Befehle;
Transfer-Befehle;
Push/Pull-Befehle.

Ein Loadbefehl ruft einen Datentransfer von einem Speicherplatz nach einem CPU-Register hervor. Das Gegenteil ist ein Storebefehl. Hier erfolgt der Datentransfer vom CPU-Register nach einem Speicherplatz.

Den Datentransfer zwischen CPU-Registern untereinander, das heißt von der CPU zum Stack (Push) und vom Stack nach der CPU (Pull), vermitteln Transportbefehle. Der erste dieser Ladebefehle, den wir behandeln, ist:

LDA Load accu M → A

Dieser Befehl steht gewöhnlich vor einer Rechen- oder logischen Aufgabe der ALU. Die erste Zahl wird in den Akku (Abb. 32a), die zweite dann bei der Bearbeitung in das Hilfsregister eingegeben. Man beachte hierzu, was im Abschnitt 9.3 bereits über die ALU geschrieben wurde.

Abhängig vom Zahlenwert, der im betreffenden Speicherplatz steht und dann in den Akku eingegeben wird, werden N und Z aus dem Statusregister 1 oder 0. Ist die Zahl negativ, wird N zu 1, sonst 0. Ist die Zahl $ 00, dann wird Z zu 1, sonst 0.

Das in der Regel auf den Ausdruck

LDA Load Accu M → A

folgende Symbol M → A gibt an, daß der Inhalt eines Speicherplatzes (M = Memory) in den Akku geladen wird. Es gibt nun noch zwei Ladebefehle:

LDX Load index x M → X
LDY Load index y M → Y

Diese Befehle stimmen mit denen für LDA überein: Auch sie haben auf N und Z des Statusregisters Einfluß, der Inhalt des Speicherplatzes geht nicht verloren.

Einer der Store-Befehle ist:

STA Store accu A → M

Nach Bearbeitung durch die ALU wird das Ergebnis in den Akku eingegeben. Falls keine weiteren Bearbeitungen vorliegen, muß dieses Ergebnis im Speicher untergebracht werden. Hierfür ist der Befehl STA vorgesehen. Dieser Befehl hat keinen Einfluß auf Statusregister und Akku-Inhalt.

Die beiden folgenden Store-Befehle sind:

STX Store index x X → M
STY Store index y Y → M

Diese Befehle stimmen mit STA vollständig überein und haben ebenfalls keinen Einfluß auf das Statusregister.

Die Transfer-Befehle haben einen Datentransport zwischen Register innerhalb der CPU zur Folge:

TAX Transfer accu to index-x A → X
TAY Transfer accu to index-y A → Y
TXA Transfer index x to accu X → A
TYA Transfer index y to accu Y → A
TSX Transfer Stack pointer to index x S → X

Der Datentransport erfolgt stets von einem Register aus, das mit „Quelle" gekennzeichnet ist (englisch Source) nach einem anderen Register, dem sog. „Empfänger" (englisch Destination). Welches Register Quelle und welches Empfänger ist, geht immer aus dem Mnemonic-Symbol hervor.

Die obenstehenden Transfer-Befehle haben auf den Inhalt von N und Z Einfluß, abhängig vom Inhalt der Quellenregister. Der Inhalt der Quellenregister geht nicht verloren. Das letzte gilt auch für den Befehl:

TXS Transfer index x to Stack pointer $X \rightarrow S$

Dieser Befehl hat jedoch keinen Einfluß auf das Statusregister.

Es wurde bereits erwähnt, daß ein Interrupt ein laufendes Programm unterbricht.

Obwohl der Befehl, mit dem der Prozessor beschäftigt ist, ausgeführt wird, bevor auf den Interrupt reagiert werden kann, ist es möglich, daß verschiedene Register im Prozessor mit Daten gefüllt sind, die nicht verlorengehen dürfen. Dies kann z.B. ein Zählerstand sein (Index x oder y), der zur Fortsetzung des Hauptprogramms noch vorhanden sein muß. Benutzt das Interrupt-Programm diese Register, dann geht deren Inhalt verloren und das Hauptprogramm kann nicht mehr auf richtige Weise fortgesetzt werden. Mit dem Push-Befehl haben wir die Möglichkeit, den Registerinhalt im Stack zu speichern:

PHA Push accu on stack $A \rightarrow M_s$

M_s bezeichnet hier einen Speicherplatz im Stack. Dieser Befehl überträgt den Inhalt in den Stack.

Es wird vorausgesetzt, daß der Befehl PHA (Opcode $ 48) auf Adresse 0200 im Speicher steht. Das Abarbeiten des Befehls durch den Prozessor verläuft nun nach folgendem Schema:

Taktzyklus	*Adressenbus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	Opcode PHA ($ 48)	Fetch Opcode	erfülle vorhergehenden Befehl. Erhöhe PC auf 0201
2.	0201	nächster Opcode ($ 8A)	Ignoriere Opcode	Decodiere PHA-Befehl, halte PC auf 0201
3.	01FF	Akku	store accu on stack	Vermindere S auf 01FE
4.	0201	nächster Opcode ($ 8A)	Fetch Opcode	erhöhe PC auf 0202

Während der ersten Taktzyklusphase Φ_1 kommt 0200 auf den Adreßbus. Dieser erste Zyklus ist ein Lesezyklus, so daß die R/$\overline{W}$-Steuerleitung „1" ist. Während der Zyklusphase Φ_2 steht der Opcode von PHA ($ 48) auf dem Datenbus. Der Opcode wird nun in das Befehlsregister eingegeben. Das sind die „externen" Prozessor-Vorgänge. „Intern"

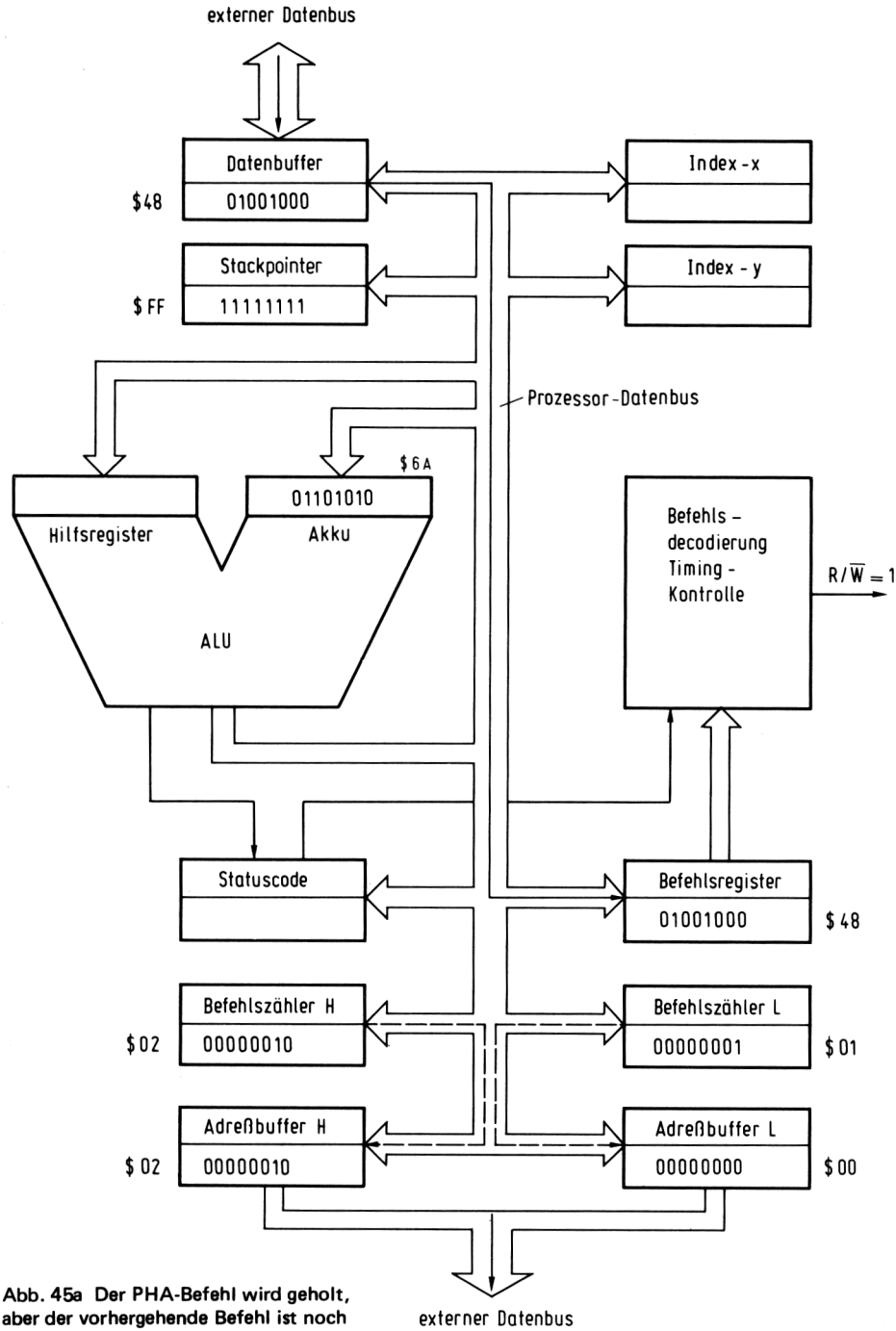


Abb. 45a Der PHA-Befehl wird geholt, aber der vorhergehende Befehl ist noch nicht abgearbeitet

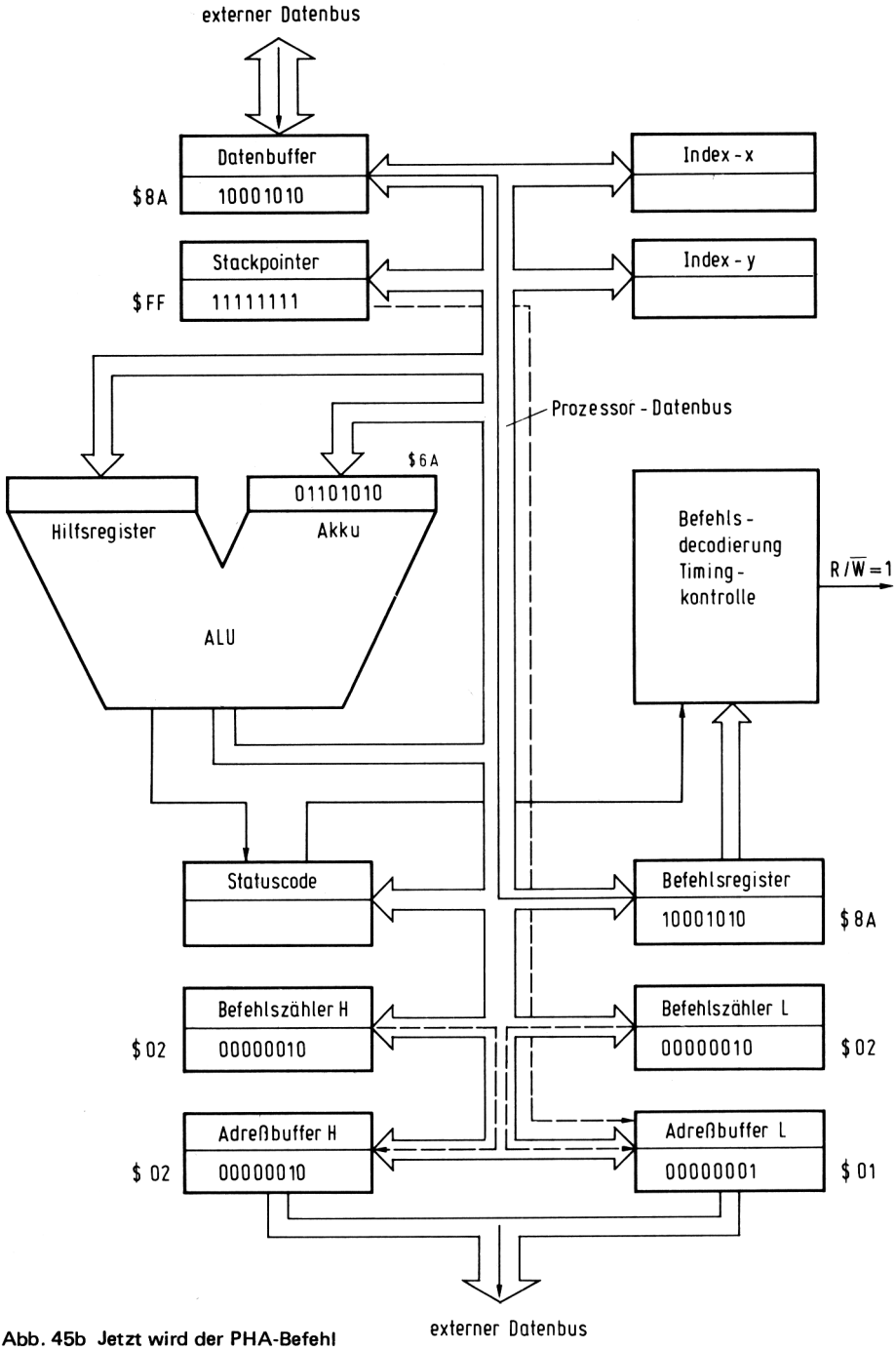


Abb. 45b Jetzt wird der PHA-Befehl decodiert

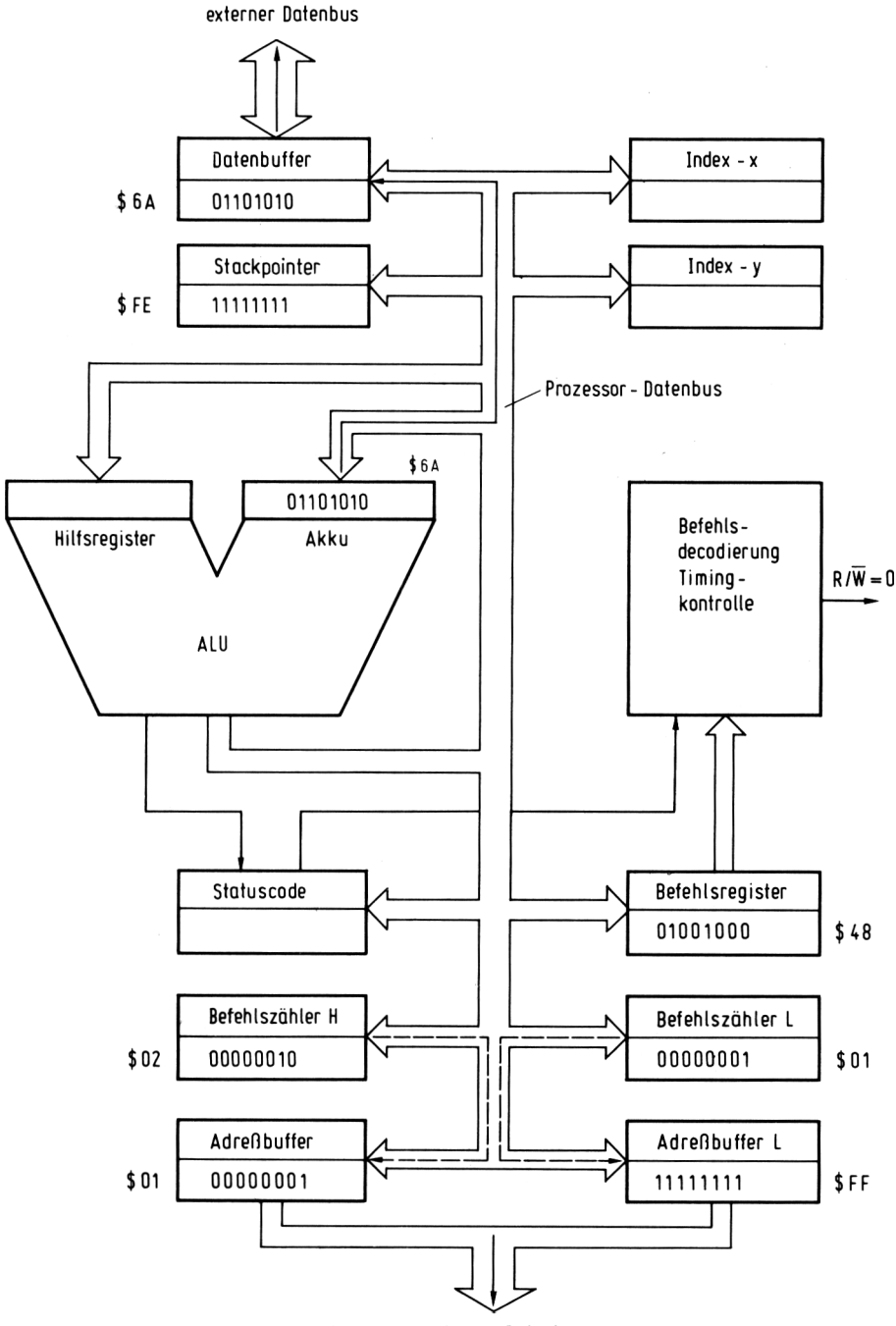


Abb. 45c Der Akku wird von PHA auf dem Stack abgelegt

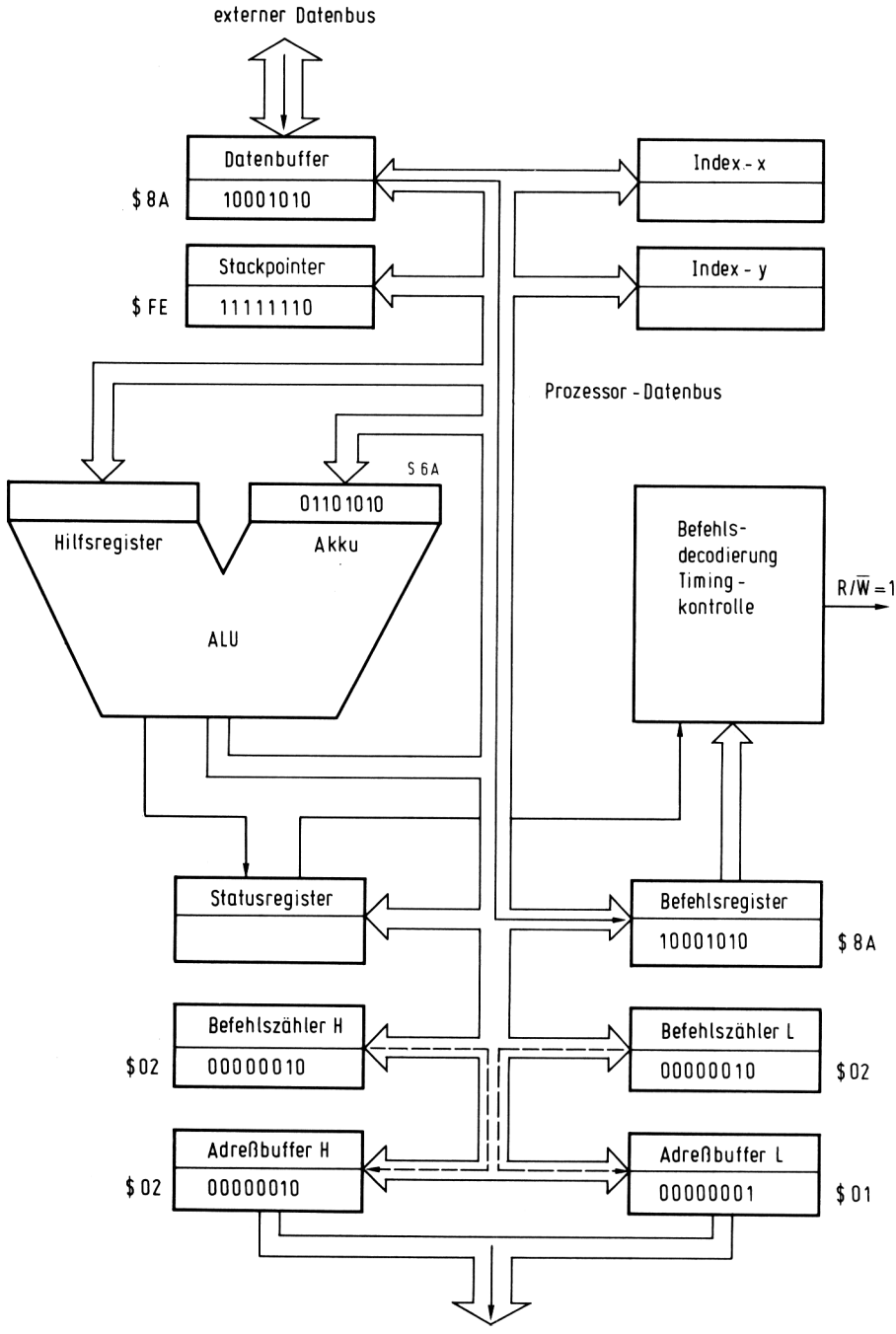


Abb. 45d Jetzt kann der nächste Befehl verarbeitet werden

findet die Fertigstellung der jeweils vorhergehenden Befehle statt. Wir nehmen an, daß hierdurch der Akku den Inhalt 6A erhält.

Welche Wirkung oder ob überhaupt ein Effekt auftritt, ist ganz vom vorhergehenden Befehl abhängig. Intern wird der Befehlszähler (PC) um 1 auf 0201 erhöht.

In *Abb. 45a* sind die Registerinhalte des Prozessors wiedergegeben. Die gestrichelten Linien in diesem Schema geben an, daß während der Phase Φ_1 des folgenden Taktzyklus der Inhalt des Befehlszählers zu den Adressenpuffern geleitet wird und damit die Adresse 0201 zugewiesen bekommt. Auf dieser Adresse steht der folgende Opcode aus dem Programm, hier $ 8A. Der zweite Taktzyklus wird jetzt durch den Prozessor völlig in Anspruch genommen, um die PHA-Instruktion zu decodieren; es kann daher kein Opcode-Fetch stattfinden. Der Inhalt der Register wird nicht verändert. Aus *Abb. 45b* geht die Situation im Prozessor hervor.

Nachdem die Befehlsform festgestellt ist, wird im dritten Taktzyklus während Φ_1 der Inhalt des Stackpointers (S) in den Adressenpuffer L geladen und der Inhalt des Adressenpuffers H wird gleich $ 01. Der Adressenbus gibt nun die Stackadresse 01FF an und während der Phase Φ_2 wird der Inhalt vom Akku auf die Adresse geschrieben. Die Steuerleitung $R/\overline{W}$ muß dafür „0" sein. Intern wird der Stackpointer auf $ FE vermindert (*Abb. 45c*).

Der vierte Zyklus stimmt mit dem ersten überein. Nun findet ein Opcode-Fetch statt; dieser Zyklus gehört dann zur folgenden Instruktion. Der Befehl PHA hat darum für die vollständige Ausführung nicht mehr als drei Taktzyklen zur Verfügung und dauert insgesamt nicht länger als 3 μs bei 1 MHz CPU-Takt (Abb. 45d).

Um auch das x- oder y-Register in den Stack zu schreiben, müssen wir einen Transfer-Befehl angeben: TXA oder TYA. Das betreffende Indexregister ist nun in den Akku geladen und kann mit PHA in den Stack überschrieben werden.

Um Akku, x- und y-Register in den Stack zu laden, müssen folgende Befehle nacheinander ausgeführt werden:

```
PHA   Akku in Stack
TXA
PHA   Index x in Stack
TYA
PHA   Index y in Stack
```

Das sind die Befehle, mit denen das Interrupt-Programm beginnen muß, um den Inhalt der betreffenden Register nicht zu verlieren. Auch das Statusregister (P) kommt in den Stack, bei einem Interrupt allerdings automatisch:

PHP Push Statusregister in Stack $P \rightarrow M_s$

Sinngemäß wird immer nach dem Laden in den Stack der Stackpointer um 1 vermindert:

$S - 1 \rightarrow S$

Am Ende des Interrupt-Programms müssen die Register aus dem Prozessor wieder in umgekehrter Reihenfolge aus dem Stack entnommen werden:

PLA Pull Accu from Stack $M_s \rightarrow A$

Nehmen wir an, daß die Instruktion PLA auf der Adresse 0300 steht, dann ist der Arbeitsablauf wie folgt:

Takt-zyklus	*Adressen-bus*	*Daten-bus*	*externer Vorgang*	*interner Vorgang*
1.	0300	Opcode PLA	Fetch Opcode	vollende vorhergehenden Befehl. Erhöhe PC auf 0301
2.	0301	nächster Opcode	negiere Opcode	decodiere PLA-Befehl, halte PC auf 0301
3.	01FE	xxx	xxx	erhöhe S auf 01FF
4.	01FF	Inhalt Akku	fetch Inhalt Akku	S bleibt 01FF
5.	0301	nächster Opcode	fetch Opcode	lade Akku

Nachdem PLA im zweiten Taktzyklus decodiert wurde, kommt der Inhalt aus dem Stackpointer auf den Adreßbus. Dieser gibt 01FE an und das ist ein „freier" Speicherplatz. Obwohl Lesezyklus, kann der Prozessor mit den Daten auf den Datenleitungen (eigenmächtig) nichts anfangen. Intern wird der Stackpointer auf 01FF erhöht. Diese Adresse kommt im vierten Zyklus auf den Adreßbus und auf diese Adresse wird der Akku-Inhalt gelagert. Der externe Vorgang akzeptiert den Datenbuswert.

Im fünften Zyklus gelangt der nächste Opcode von der Adresse 0301 auf den Datenbus. Der externe Vorgang ist der Opcode-Fetch. Intern wird der Akku mit dem aus dem Stack geholten Inhalt geladen. Da im fünften Taktzyklus schon der Opcode-Fetch der nächsten Instruktion stattfindet, dauert PLA insgesamt vier Taktzyklen.

Das Laden aller Register des Prozessors aus dem Stack verläuft mit folgenden Befehlen:

```
PLA
TAY   Index y geladen aus Stack
PLA
TAX   Index x geladen aus Stack
PLA   Akku geladen aus Stack
```

Hier ist besonders auf die Reihenfolge im Zusammenhang mit „Last in, first out" zu achten. Auch das Statusregister kann aus dem Stack geladen werden:

PLP Pull Statusregister aus Stack $M_S \rightarrow P$

Durch PLP soll zuerst der Stackpointer erhöht werden: S + 1 → S. Danach wird der Inhalt in das Statusregister zurückgesetzt.

Von den Push- und Pull-Instruktionen hat nur PLA Einfluß auf N und Z des Statusregisters. Muß der Inhalt aller Register des Prozessors aufbewahrt werden, z.B. für ein Unterprogramm, dann muß zuerst der Inhalt des Statusregisters in den Stack geladen werden:

```
PHP
PHA
TXA
PHA
TYA
PHA
```

Nun wieder zurück an das Ende des Unterprogramms:

```
PLA
TAY
PLA
TAX
PLA
PLP
```

Das Statusregister wird nun als letztes aus dem Stack geholt, so daß sein Inhalt nicht mehr durch andere Befehle geändert werden kann (z.B. durch einen Transfer-Befehl oder durch PLA).

10.3 Rechenbefehle

Die arithmetischen oder Rechenbefehle lassen sich in zwei Gruppen einteilen:

a) Addieren oder Subtrahieren von zwei Zahlen
b) Anheben oder Vermindern eines Registerinhaltes um 1.

Beim Addieren von zwei Zahlen spielt das Carry-Bit eine wesentliche Rolle. Wie bereits bei der binären Addition im Abschnitt 4.1 erwähnt, wird das Carry-Bit dann benötigt, wenn das Ergebnis einer Addition für ein Register zu groß ist. Der Wert eines Carry-Bit ist auch bei der Addition von zwei Zahlen erforderlich, die größer als acht Bit sind. Für die Addition gibt es nur einen Befehl:

ADC Add memory to accu with carry $A + M + C \rightarrow A$

Dieser Vorgang hat auf die Werte N, Z, C und V des Statusregisters Einfluß:

N = 1, Ergebnis negativ
Z = 1, Ergebnis Null
C = 1, Ergebnis zu groß für das Register
V = 1, Der Inhalt von N gibt den Zustand nicht richtig wieder.

Diese Bedingungen sind in vorhergehenden Abschnitten bereits behandelt worden. Das halfcarry H mancher Prozessortypen fehlt in der CPU 6502; es ist wegen des SED-Befehls überflüssig.

Im nächsten Beispiel sind zwei positive 8-Bit-Zahlen zu addieren:

	binär	*dezimal*
	C 0	
	a 00011011	37
	M 01110111+	119+
a+M+C	10010010	146

C=0 Z=0 N=1 V=1

Das achte Bit dieser Addition ist 1 (B7), daher N = 1; das Ergebnis ist nicht Null, deshalb Z = 0. Da dies eine Addition von zwei positiven Zahlen betrifft, N aber 1 wird, eine unrichtige Bedingung, ist V = 1.

In diesem Beispiel muß das Carry-Bit zuerst auf 0 gesetzt werden. Der Inhalt von C ist vor der Addition völlig willkürlich und könnte darum auch 1 sein. Das Carry-Bit wird Null mit folgendem Befehl:

CLC Clear carry flag C = 0

Eine Addition kann sowohl binär als auch im BCD-Code durchgeführt werden. Die Addition im Beispiel wird binär ausgeführt. Flag D im Statusregister muß Null sein. Auch der Inhalt von D kann willkürlich sein und muß deshalb den Wert Null erhalten. Hierzu der Befehl:

CLD Clear decimal mode D = 0

Es ergibt sich also nachfolgende Reihe von Befehlen:

```
CLD
CLC
LDA  Adresse  a
ADC  Adresse  M
STA  Adresse  Summe.
```

Nach CLD und CLC wird zuerst a in den Akku geladen (Abb. 32a). Nun folgt der Arbeitsbefehl ADC. Dieser hat zur Folge, daß M in das Hilfsregister geladen wird (Abb. 32b) und die Addition in die ALU überführt wird, woraus sich dann das Ergebnis im Akku ergibt (Abb. 32c). Die zweite Zahl ist dann M.

Der Akku-Inhalt wird mit dem Befehl STA auf den hierfür bestimmten Speicherplatz überführt.

Die Addition von zwei 16-Bit-Zahlen muß in zwei Schritten durchgeführt werden, da der Prozessor nur 8-Bit-Zahlen addieren kann. Im folgenden Beispiel werden zwei positive Zahlen addiert:

	H	L
a	00111100	10100010
M	01010101	10111100+
Summe	10010010	01000000

C=0 Z=0 N=1 V=1

Die Addition durch den Prozessor läuft wie folgt ab:

```
1. Schritt  C              0
            a L     10100010
            M L     10111100
                    --------
            Summe L 01011110

            C=1  Z=0  N=0  V=1
```

Der Prozessor glaubt es mit zwei negativen Zahlen zu tun zu haben, aber B7 des Ergebnisses ist 0, deshalb wird V = 1.

```
2. Schritt  C              1
            a H     00111100
            M H     01010101+
                    --------
            Summe H 10010010   C=0  Z=0  N=1  V=1
            Summe   10010010   01011110
```

Der Inhalt von C, N und V vom zweiten Schritt bestimmt die „gesamte" Summe. Das gilt aber nicht für Z, wie aus dem nächsten Beispiel ersichtlich, in dem eine negative Zahl zu einer positiven addiert werden soll.

```
            a       00111100   10100010
            M       11000011   10111100+
                    --------   --------
            Summe   00000000   01011110  C=1  V=1  Z=0   N=0  V=0

1. Schritt  C              0
            aL      10100010
            ML      10111100+
                    --------
            Summe L 01011110   C=1  Z=0  N=0  V=1
2. Schritt  C              1 ←┘
            aH      00111100
            MH      11000011+
                    --------
            Summe H 00000000   C=1  Z=1  N=0  V=0
            Summe   00000000   01011110
```

Beim zweiten Schritt wird Z „1". Das stimmt natürlich nur für die zweite Teilsumme.

Als letztes ein Beispiel für eine Addition von zwei negativen Zahlen.

```
            a       11101101   11101011
            M       11101110   11001110+
                    --------   --------
            Summe   11011100   10111001

            C=1  Z=0  N=1  V=0
```

1. Schritt	C	0				
	a L	11101011				
	M L	11001110+				
	Summe L	10111001	C=1	Z=0	N=1	V=0
2. Schritt	C	1 ←				
	a H	11101101				
	M H	11101110+				
	Summe H	11011100	C=1	Z=0	N=1	V=0
	Summe	11011100	10111001			

Folgende Befehle sind für diese Addition erforderlich:

```
            CLD
            CLC
            { LDA  Adresse  a L
1. Schritt  { ADC  Adresse  M L
            { STA  Summenadresse L
            { LDA  Adresse  a H
2. Schritt  { ADC  Adresse  M H
            { STA  Summenadresse H
```

Der Befehl CLC kommt nur beim ersten Schritt vor. Der Wert von C für den zweiten Schritt wird durch den ersten Schritt bestimmt.

Die Addition im BCD-Code stimmt mit der Binär-Addition überein. Die erforderlichen Korrekturen werden automatisch durchgeführt, sobald D = 1 ist. An die Stelle von CLD tritt der Befehl SED. (Er ist allerdings nur für die Befehle ADC und SBC relevant.)

SED Set decimal mode D = 1

Im übrigen sind die Befehle und die Reihenfolge ebenso wie beim binären Rechnen. Beim Subtrahieren wird nur der Subtrahend in den Akku geladen. Der Subtrahend ist die Zahl M. Die hierbei erforderliche Instruktion ist:

SBC Subtract memory from accu with borrow $A - M - \overline{C} \rightarrow A$

Dieser Vorgang hat auf den Inhalt von C, Z, N und V des Statusregisters Einfluß. Der Vorgang ist mit dem Komplementsystem zu erklären. Von M wird das Zweierkomplement errechnet. Hierzu wird zuerst von M die Inversion ermittelt und danach 1 addiert. Dazu ist das Carry-Bit erforderlich, das deshalb nicht Null, sondern 1 sein muß, und hieraus folgt $A - M - \overline{C} \rightarrow A$.

Es folgt nun eine Reihe von Beispielen für die Subtraktion von 8-Bit-Zahlen. Zuerst eine Subtraktion von zwei positiven Zahlen mit einem positiven Ergebnis:

	a	01101001	M 01001100			
	C	1				
	a	01101001				
	$\overline{M}$	10110011+				
a–M–	$\overline{C}$	00011101	C=1	Z=0	N=0	V=0

Subtraktion von zwei positiven Zahlen mit negativem Ergebnis:

```
        a       01001100    M 01101001
        C             1
        a       01001100
        M̄       10010110+
a–M– C̄          11100011    C=0  Z=0  N=1  V=0
```

Eine positive Zahl von einer negativen subtrahieren:

```
        a       10110110    M 01011001
        C             1
        a       10110110
        M̄       10100110+
a–M– C̄          01011101    C=1  Z=0  N=0  V=1
```

In diesem Ergebnis ist Bit 7 Null, daher N = 0. In dieser Subtraktion ist das Ergebnis negativ (eine Addition von zwei negativen Zahlen), daher V = 1.

Eine negative Zahl von einer positiven subtrahieren:

```
                01011001    M 10110110
        C             1
        a       01011001
        M̄       01001001+
a–M– C̄          10100011    C=0  Z=0  N=1  V=1
```

Der Rechenvorgang geht auf eine Addition von zwei positiven Zahlen zurück. N = 1 (B7 = 1), hieraus V = 1. Das Ergebnis ist eine positive Zahl.

Eine negative Zahl von einer negativen Zahl subtrahieren, mit positivem Ergebnis:

```
        a       11010100    M 10010010
        C             1
        a       11010100
        M̄       01101101+
a–M– C̄          01000010    C=1  Z=0  N=0  V=0
```

Das für diese Subtraktion erforderliche Unterprogramm läuft auf gleiche Art ab wie für die Addition:

```
CLD
SEC
LDA  Adresse a
SBC  Adresse M
STA  unterschiedliche Adressen.
```

Der zweite Befehl ist SEC anstelle von CLC. Das Carry-Bit muß für das Rechnen im Zweierkomplement 1 sein.

SEC Set carry flag C = 1

Die Subtraktion zweier 16-Bit-Zahlen erfolgt ebenfalls in zwei Schritten:

```
        a        01101010   00011101
        M        01001011   01101011

        C               1
        a L      00011101
        M̄ L      10010100+
                 ---------
Differenz L      10110010   C=0  Z=0  N=1  V=0
                            ┌──┘
        C               0 ←┘
        a H      01101010
        M̄ H      10110100+
                 ---------
Differenz H      00011110   C=1  Z=0  N=0  V=0

Differenz        00011110   10110010
```

Die zweite Gruppe von Rechenbefehlen betrifft das Anheben oder Vermindern eines Zählers.

Für ein einfaches Anheben eines Zählers um „1" gelten folgende Befehle:

INX	Increment index-x by one	x + 1 → x
INY	Increment index-y by one	y + 1 → y

Diese Befehle beeinflussen den Inhalt des Statusregisters N und Z, aber nicht C und V. Das ist auch bei dem Befehl

INC	Increment memory by one	M + 1 → M

der Fall. Dieser Befehl ermöglicht den Einsatz jedes beliebigen Speicherplatzes als Zähler. Mit den vorhergehenden Instruktionen ist dies jedoch nicht der Fall. Diese beziehen sich ausschließlich auf das betreffende Index-Register.

Für das Vermindern von Zählern gelten folgende Befehle:

DEX	Decrement index-x by one	$x - 1 \rightarrow x$
DEY	Decrement index-y by one	$y - 1 \rightarrow y$
DEC	Decrement memory by one	$M - 1 \rightarrow M$

Auch jetzt wird der Inhalt von N und Z beeinflußt, jedoch nicht der Inhalt von C und V.

10.4 Vergleichsbefehle

Die Vergleichsbefehle sind im Prinzip Rechenbefehle. Ein Vergleichsbefehl führt eine Subtraktion aus. Die ursprünglichen Zahlen gehen dabei jedoch nicht verloren, da das Resultat nicht in den Akku geladen wird. Die Größe des Ergebnisses wird nicht angegeben. Der Befehl beeinflußt lediglich den Inhalt von N, Z und C im Statusregister. Die Vergleichsbefehle geben eine „Bedingung" an für einen bedingten Sprungbefehl. Es gibt hier nun drei Möglichkeiten:

a) kleiner als
b) größer als
c) gleich.

Die zur Verfügung stehenden Befehle beziehen sich auf Akku, Index-x-und Index-y-Register. Die zweite Zahl steht immer auf einem Speicherplatz (M).

CMP	Compare memory and accu	$A - M$
CPX	Compare memory and index x	$X - M$
CPY	Compare memory and index y	$Y - M$

Wir untersuchen die drei Möglichkeiten:

a) Zahl a ist kleiner als M: $a < M$
Die Zahlen a und M sind beide positiv:

a=01011001 M=01101011

```
a   01011001
M   10010100
C          1+
    ---------
    11101110  C=0,  N=1,  Z=0,  V=0
```

Dasselbe Ergebnis der Zustandsflags erhält man, wenn beide Zahlen negativ sind.
Die Zahlen a und M haben entgegengesetzte Vorzeichen:

a ist negativ M ist positiv
a=10010010 M=01010010

```
a     10010010
M     10101101
C           1+
      --------
      01000000  C=1, N=0, Z=0, V=1
```

a < M: C=0, N=1, Z=0, V=0 of (p)
C=1, N=0, Z=0, V=1 (q).

Sowohl in (p) als auch in (q) sind N und L verschieden, also N∀V = 1.

b) Zahl a ist größer als Zahl M: a > M
Die Zahlen a und M sind beide positiv:

a=01101011 M=01011001

```
a     01101011
M     10100110
C           1+
      --------
      00010010  C=1, N=0, Z=0, V=0
```

Dasselbe Ergebnis der Zustandsflags erhält man, wenn beide Zahlen negativ sind.
Die Zahlen a und M haben entgegengesetztes Vorzeichen.

a positiv M negativ
a=01011001 M=10110110

```
a     01011001
M     01001001
C           1+
      --------
      10100011  C=0, N=1, Z=0, V=1
```

a > M: C=1, N=0, Z=0, V=0 of (r)
C=0, N=1, Z=0, V=1 (s)

Sowohl in (r) als auch in (s) sind N und V gleich, also N∀V = 0

c) Zahl a ist gleich Zahl M:

a=M=01101011

```
a     01101011
M     10010100
C           1+
      --------
      00000000  C=1, N=0, Z=1, V=0
```

Auch in diesem Fall ist N gleich V, also $N \forall V = 0$. Für die Zahlen a und M gibt es fünf Möglichkeiten:

$a > M$, $a \geqslant M$, $a = M$, $a \leqslant M$, $a < M$

Im Zusammenhang mit den Zustandsflags sind folgende Bedingungen zu stellen:

$a > M$: $Z \vee (N \forall V) = 0$
$a \geqslant M$: $N \forall V = 0$
$a = M$: $Z = 1$
$a \leqslant M$: $Z \vee (N \forall V) = 1$
$a < M$: $N \forall V = 1$

10.5 Logische Befehle

Drei logische Vorgänge kann der Prozessor ausführen, AND ($\wedge$), OR ($\vee$) und Exklusiv-OR ($\forall$). Es sind dies die Befehle:

AND	And Memory with accu	$A \wedge M \rightarrow A$
ORA	Or Memory with accu	$A \vee M \rightarrow A$
EOR	Exclusive-or memory with accu	$A \forall M \rightarrow A$

Diese Befehle beeinflussen N und Z im Statusregister. Das Ergebnis wird in den Akku überschrieben.

Der Computer wendet die Befehle in zwei gleichen Bitworten an und arbeitet so die Blöcke Bit für Bit ab. Die Beispiele logischer Verarbeitungen aus Abschnitt 7.2. sind hier vollständig anwendbar. Es ist auch möglich, einen Bitwert durch Maskieren zu bestimmen, eine Tetrade zu isolieren, Tetraden zusammenzufügen und die Inversion einer Zahl zu bilden. Gesetzt den Fall, der Wert des dritten Bits einer Zahl soll bestimmt werden:

a	01001010	
Maske b	00000100	
$a \wedge b$	00000000	$Z = 1$

Die Reihenfolge der Befehle hierfür muß lauten:

```
LDA $ 04,
AND Adresse a
Bestimme Inhalt von Z
```

Mit dem ersten Befehl wird die Maske in den Akku geladen:

$\$ 04 = 00000100_{(2)}$

Mit dem zweiten Befehl führt der Prozessor den AND-Auftrag zwischen der Maske im Akku und der Zahl a aus. Das Ergebnis steht dann im Akku. Dann kann der Inhalt von Z aus dem Statusregister bestimmt werden.

Durch das Laden des Ergebnisses in den Akku ist die Maske verlorengegangen. Dies kann nachteilig sein, sofern für mehrere Zahlen der Wert des dritten Bit bestimmt werden muß. Folgende Befehle müssen ausgeführt werden:

```
LDA $ 04
AND Adresse a
Bestimme Inhalt von Z,
LDA $ 04
AND Adresse b
Bestimme Inhalt von Z,
usw.
```

Der Befehl LDA $ 04 muß immer wiederholt werden. Es gibt einen Befehl, mit dem der Prozessor den AND-Vorgang ausführt, ohne daß der Inhalt des Akkus geändert wird, den sog. BIT-Befehl.

BIT Test bits in memory with accu A ∧ M.

Mit diesem Befehl wird nicht nur der Wert des maskierten Bit, sondern auch der Wert von Bit 7 und Bit 6 bestimmt. Der Befehl hat auch gleichzeitig Einfluß auf das Statusregister, und zwar daß der Wert von N gleich ist mit dem Wert von B7 des untersuchten Wortes, der Wert von V gleich ist mit dem Wert von B6 desselben untersuchten Wortes und der Wert Z vom Vorgang A ∧ M abhängig ist, ohne daß sich der Inhalt des Akku dabei ändert. Es wird dabei gleichzeitig der Wert von drei Bit bestimmt, und zwar von B7, B6 und vom maskierten Bit (wenn nur eine 1 in der Maske steht).

```
a          01001010
Maske b    00000100
           --------
a∧b        00000000  N=0  V=1  Z=1

a          10101101
Maske b    00000100
           --------
A∧b        00000100  N=1  V=0  Z=0
```

Die Befehlsreihe zum Maskieren mehrerer Zahlen ergibt sich wie folgt:

```
LDA $ 04,
BIT, Adresse a,
Bestimme Inhalt von Z,
BIT, Adresse b,
Bestimme Inhalt von Z,
usw.
```

Das ständige Neuladen des Akku mit $ 04 entfällt hierbei.

Da es möglich ist, den Wert eines bestimmten Bits aus einem Datenwort zu ermitteln, muß es auch möglich sein, einem bestimmten Bit aus einem Datenwort einen Wert zu geben. Es soll in einem Wort das dritte Bit Null werden:

a	01011x01	hier ist x 0 oder 1
Maske M	11111011	
a∧M	01011001	

Da das dritte Bit der Maske Null ist und die Betriebsart AND angewandt wird, ist das dritte Bit 0.
Die Befehlsfolge ergibt sich somit zu:

LDA $ FB ($ FB = 11111011_2)
AND Adresse a,
STA Adresse a.

Durch den Befehl LDA wird die Maske in den Akku geladen. Nach dem AND-Befehl ist der Inhalt des Akku gleich dem Ergebnis und damit Bit drei 0. Daraufhin wird dieses Ergebnis in den Speicherplatz der Zahl a eingegeben. Hier finden wir folglich die ursprüngliche Zahl mit dem dritten Bit 0.

Dem Bit kann auch der Wert 1 zugeordnet werden:

a	01011x01	x kann 1 oder 0 sein
Maske M	00000100	
a∨M	01011101	

Hier ist nun ein OR-Befehl eingesetzt.
Die Reihenfolge der Befehle ergibt sich dann:

LDA $ 04
ORA Adresse a
STA Adresse a

10.6 Schiebebefehle

Die Schiebebefehle verlagern jedes Bit von einem Register eine Stelle nach rechts oder links, abhängig von dem betreffenden Befehl. Hierbei geht kein einziges Bit verloren. Auch das aus dem Register geschobene Bit bleibt erhalten und wird ständig im Statusregister auf C gespeichert. Wird an einer Seite des Registers ein Bit herausgeschoben, wird an der anderen Seite ein Bit eingeschoben. Dies kann sowohl eine „Null" sein oder auch der Inhalt von C vor dem Schiebevorgang.

Es gibt vier Schiebebefehle:

ROR	Rotate right
ROL	Rotate left
LSR	Logical shift right
ASL	Arithmetic shift left

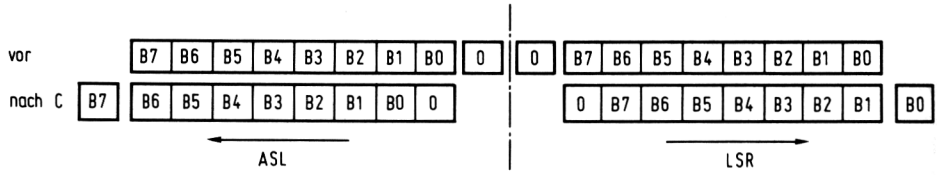
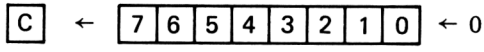


Abb. 46 Registerzustand vor und nach einem Links-Schiebebefehl

Abb. 46 gibt den Registerzustand vor und nach einem „Schiebebefehl" wieder. Durch ASL wird B0 mit „0" geladen und B7 befindet sich auf C. Alle anderen Bits wandern eine Stelle nach links. Durch LSR wird Bit 7 mit Null geladen und Bit 0 mit dem vorherigen Carry-Zustand. Diese Befehle sind einander entgegengesetzt.

Das Symbol für ASL ist:



Hier soll der Abschnitt 4.3. in Erinnerung gebracht werden, betreffend die Multiplikation mit $2_{(10)}$:

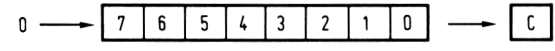
Schiebt man eine Binärzahl in einem Register eine Stelle nach links, dann wird diese mit 2 multipliziert.

Damit wird das niederwertigste Bit (Bit 0) Null. Dieser Vorgang stimmt völlig mit dem Befehl ASL überein.

Der Befehl LSR ist entgegengesetzt zu ASL und teilt die Binärzahl durch 2:

Schiebt man eine Binärzahl in einem Register einen Platz nach rechts, dann wird diese durch 2 dividiert.

Nun wird das höchste Bit (B7) „0". Das Symbol für LSR ist:



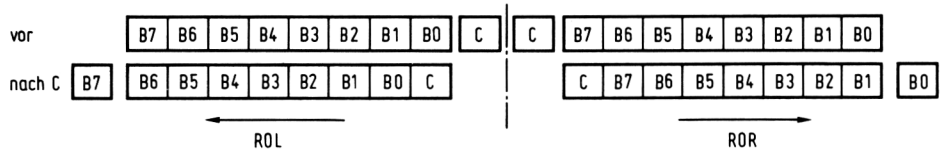
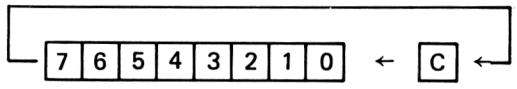
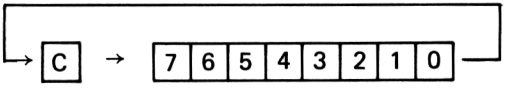


Abb. 47 Registerzustand vor und nach einem Links-Rotierbefehl

In *Abb. 47* ist der Registerzustand vor und nach einem „Rotier"-Befehl dargestellt. Sowohl bei ROR als auch bei ROL ist der Übertrag in den Kreislauf einbezogen. Der Kreis ist „geschlossen", wenn C zuerst in das Register und später wieder aus dem Register geschoben wird. Das Symbol für ROL ist:



und das Symbol für ROR:



10.7 Bedingte Sprungbefehle (Branch)

Wenn der Prozessor ein Programm abarbeitet, soll der Befehlszähler regelmäßig um 1 erhöht werden und folglich auf eine Adresse hinweisen, die um 1 höher ist als die vorhergehende. Trifft der Prozessor in diesem Programm auf einen Sprungbefehl, dann erhält der Befehlszähler einen neuen Inhalt, um mit diesem als erste Adresse ein Programmstück zu durchlaufen, das an anderer Stelle des Speichers abgelegt ist. Der Befehlszähler wird dann wieder um 1 erhöht, so daß die Adressen in numerischer Reihenfolge weiter abgearbeitet werden können.

Bei einem bedingten Sprungbefehl wird der Sprung nur dann ausgeführt, wenn die Bedingung erfüllt ist. Ist dies nicht der Fall, dann wird der Befehlszähler für die nächste Adresse normal um 1 erhöht. Die Bedingungen für die Sprünge beziehen sich immer auf den Inhalt eines Bits im Statusregister, etwa so:

„Springe, wenn Z = 1"

oder auch

„Springe, wenn Z = 0"

Bei bedingten Sprungbefehlen bestimmen die Inhalte von C, Z, N und V, ob der Sprung ausgeführt wird oder nicht. Es gibt acht verschiedene bedingte Sprungbefehle:

BCC	Branch on carry clear	Branch on C=0
BCS	Branch on carry set	Branch on C=1
BNE	Branch on result not equal	Branch on Z=0
BEQ	Branch on result equal	Branch on Z=1
BPL	Branch on result plus	Branch on N=0
BMI	Branch on result minus	Branch on N=1
BVC	Branch on overflow clear	Branch on V=0
BVS	Branch on overflow set	Branch on V=1

Die Sprungbefehle (Branches) verändern den Inhalt des Statusregisters nicht. Der Operationscode des betreffenden Sprungbefehls muß noch eine Zahl erhalten, aus der hervorgeht, um wieviele Speicherplätze nach vorn oder zurück gesprungen werden soll.

In Abb. 7 ist ein sehr oft vorkommender bedingter Sprung dargestellt. Hier wird nach „Loop" zurückgesprungen, solange ein Zähler (x) noch nicht Null ist. Es gibt jedoch kei-

nen Sprungbefehl, der den Inhalt eines Zählers als Bedingung erkennt. Dem Sprungbefehl muß direkt ein Auftrag vorausgehen, der N, Z, C oder V auf „0" oder „1" legt. Das ist in dem erwähnten Beispiel tatsächlich der Fall.

Nehmen wir an, daß mit x das Index-Register x gemeint ist, dann wird gerechnet:

$x - 1 \rightarrow x$

ausgeführt durch den DEX-Befehl. Dieser Befehl soll Z auf „1" legen, sobald nach einer Verminderung der Inhalt des Index-x-Registers Null ist. Es muß nach „Loop" gesprungen werden, wenn das Resultat nicht Null ist, d.h., bei Z = 0. Dazu dient der Befehl BNE.

Die letzten Programmbefehle aus Abb. 7 lauten dann:

```
DEX
BNE n
```

Die Zahl n gibt hier an, um wieviele Speicherplätze vor oder zurück gegangen werden soll. Bei einem Vorwärtssprung ist n positiv, bei einem Rückwärtssprung, wie in diesem Beispiel, muß n negativ sein.

Im vorhergehenden wurde festgestellt, daß die Sprungbedingung vom Ergebnis eines Vorganges abhängt. Es ist aber auch möglich, die Sprungbedingung von der Größe eines Speicherinhalts abhängig zu machen. Hier kommen die Vergleichsbefehle zur Anwendung (CMP, CPX und CPY). Nehmen wir an, a ist eine Zahl in einem Register (z.B. Akku oder Index-Register) und M eine Vergleichszahl (z.B. 5). Dann sind fünf Vergleichsmöglichkeiten gegeben:

$a > M \quad a \geqslant M \quad a = M \quad a \leqslant M \quad a < M$

Nach Abschnitt 10.4 haben nach einem CMP-Befehl (a im Akku und M auf einem Speicherplatz, CMP a– M) bei diesen fünf Möglichkeiten Z, N und V folgende Inhalte:

$a > M$	$Z \vee (N \veebar V) = 0$
$a \geqslant M$	$N \veebar V = 0$
$a = M$	$Z = 1$
$a \leqslant M$	$Z \vee (N \veebar V) = 1$
$a < M$	$N \veebar V = 1$

Wir nehmen an, daß ein Sprung unter der Bedingung $a < M$ durchgeführt werden soll. Für die Zustandsflags gilt dann $N \veebar V = 1$. Hierzu die Wahrheitstabelle:

N	*V*	$N \veebar V$
0	0	0
0	1	1
1	0	1
1	1	0

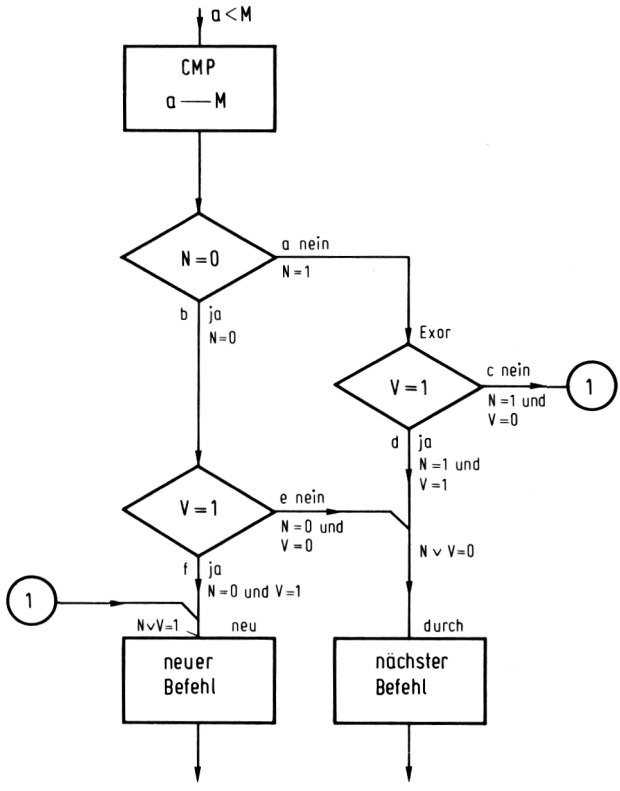


Abb. 48 Flußdiagramm zur Auswertung der N-und V-Statusflags

Nur die Zeilen zwei und drei erfüllen die Sprungbedingung. In diesem Falle erfolgt ein Sprung nach einem (neuen) Unterprogramm, das sich in einem anderen Speicherteil befindet.

Das Flußdiagramm zur Ausführung der Bedingung N∀V = 1 ist in *Abb. 48* dargestellt. Dieses Flußdiagramm hat zwei senkrechte Äste. Die Bedingungen N = 0 und V = 1 sind so gewählt, daß der rechte Ast an den Befehl des folgenden Unterprogramms anschließt (mit dem Kennzeichen „durch" von „durchgehen"). Das ist für das Aufstellen der Befehlsreihe einfacher. Der linke Ast endet mit dem Sprung auf das neue Unterprogramm.

Nach dem CMP-Befehl folgt die Bedingung N = 0. Wird diese nicht erfüllt, dann folgt ein Sprung nach dem Befehl mit dem Kennzeichen „Exor", das ist in Richtung a (N = 1). Wird die Bedingung hier erfüllt (Richtung b, N = 0), folgt im nachfolgenden Befehl die Bedingung V = 1. In Richtung e besteht auch keine Möglichkeit, und es findet ein Sprung nach „durch" statt, da N = 0 und V = 0 (Zeile 1 der Wahrheitstabelle). In dieser Richtung gilt nämlich N = 0 wie auch V = 1 und somit wird die allgemeine Sprungbedingung erfüllt (Zeile 2 der Wahrheitstabelle). Die Sprungadresse wird hier mit „neu" bezeichnet.

Unter dem Kennzeichen „Exor" steht die Bedingung V = 1. Die Richtung c ist hier nicht geeignet, da sowohl N = 1 also auch V = 0 sind, deshalb muß nach „neu" gesprun-

gen werden (Zeile 3 der Wahrheitstabelle). Für Richtung d gilt N = 1 und V = 1, so daß nach Zeile 4 der Wahrheitstabelle kein Sprung stattfinden kann. Das Abarbeiten des Programms erfolgt nun wie gewohnt. Die Reihenfolge der Befehle ist aus untenstehender Tabelle ersichtlich:

Kennzeichen	*Operationsadresse*
a < M	LDA Adresse a
	CMP Adresse M
	BMI +4 Exor
	BVC +4 durch
	BVS +40 neu
Exor	BVC +38 neu
durch	nächster Befehl

Gemäß dieser Tabelle wird zuerst der linke vertikale Ast abgearbeitet. Der LDA-Befehl bringt a in den Akku. Der nachfolgende CMP-Befehl hat keinen Einfluß auf den Akku-Inhalt, gibt aber die entsprechenden Werte an C, N, Z und V aus dem Statusregister in Abhängigkeit vom Ergebnis aus a–M. Nun muß, wenn N=1, ein Sprung folgen, und zwar mit Befehl BMI. Die Zahl +4 hinter diesem Befehl deutet darauf hin, daß vier Stellen übersprungen werden müssen nach dem Befehl für „EXOR". Es sind nämlich zwei Speicherplätze für jeden Verzweigungsbefehl erforderlich. Der nächste Befehl hat einen Sprung, da V = 0, nach „durch" zur Folge. Auch hier müssen wieder vier Stellen übersprungen werden. Auch für V = 1 ergibt sich ein Sprung, und zwar auf „neu". Nun ist die allgemeine Bedingung N∀V = 1 erfüllt. Es wird 40 Stellen weiter gesprungen, nach „neu", wo sich der neue Befehl befindet. Die Anzahl von 40 Stellen ist im übrigen völlig beliebig gewählt worden.

Nachdem wir nun den linken vertikalen Zweig abgearbeitet haben, wenden wir uns dem rechten vertikalen zu. Da V = 0, muß wiederum ein Sprung ausgeführt werden, deshalb BVC. Auch hier muß nach „neu" gesprungen werden. Wir sind durch diesen Befehl um zwei Speicherplätze in Richtung „neu" gekommen, so daß nach BVC die Zahl +38 erscheint. Der nächste Befehl hat das Kennzeichen „durch". Der nächste Programmteil schließt, wie gewöhnlich, hier an.

In Abb. 48 sind zwei „Äste" zu erkennen, einer nach „neu" mit dem Ergebnis N∀V=1 und einer nach „durch" mit dem Ergebnis N∀V = 0. Hiervon ist die Sprungbedingung Z (N∀V) = 1 (a ⩽ M), abgeleitet. Die Wahrheitstabelle ergibt sich wie folgt:

N	*V*	*Z*	*N∀V*	*ZV(N∀V)*
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	1

Mit diesen Wahrheitstabellen lassen sich Zeile für Zeile die logischen Funktionen Λ, V oder ∀ ermitteln.

Das Flußdiagramm geht aus *Abb. 49* hervor, in dem das Diagramm aus Abb. 48 eingearbeitet ist. Da N∀V = 1 muß unabhängig von Z stets nach „neu" gesprungen werden. Wenn N∀V = 0, kann nur nach „neu" gesprungen werden, wenn Z = 1 ist. Diese Bedingung wird durch Zufügung von Block Z = 0 erfüllt. Die Befehlsfolge ergibt sich somit zu:

Label	*Operation*
a ⩽ M	LDA Adresse a
	CMP Adresse M
	BMI + 4 Exor
	BVC + 4 Oder
	BVS + 40 neu
Exor	BVC + 38 neu
Oder	BEQ + 36 neu
durch	nächster Befehl

Auf den Befehl mit Kennzeichen „Exor" folgt, wenn Z = 1, die Sprungadresse BEQ. Die Zahlen hinter den Verzweigungsbefehlen sind nun besonders zu beachten. Die be-

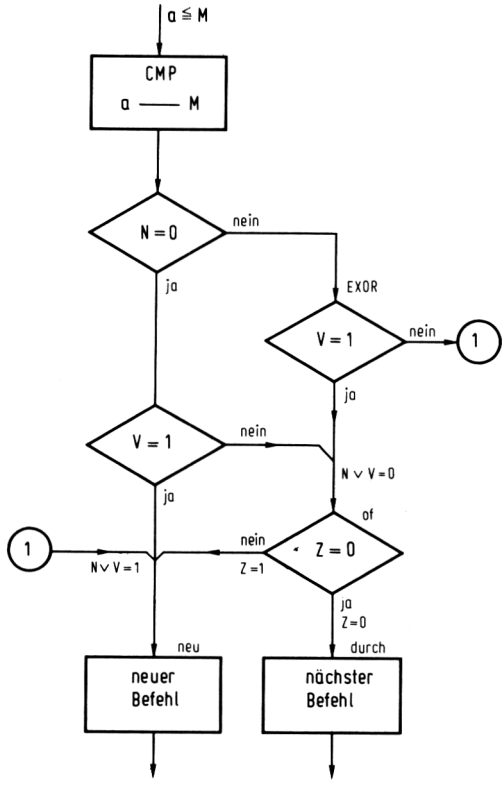


Abb. 49 Erweitertes Flußdiagramm nach Abb. 48

dingten Sprünge, d.h. N∀V = 0 und ZV (N∀V) = 0 für A≥M bzw. a>M sind auf ähnliche Weise zu verwirklichen. Der bedingte Sprung für Z = 1 (a = M) ist einfach zu realisieren und bedarf keiner weiteren Erklärung.

Neben den fünf genannten Möglichkeiten, die beim Vergleich von zwei Zahlen (a–M) auftreten, gibt es auch noch andere Voraussetzungen, unter denen ein Sprung stattfinden kann. Eine derartige Anwendung eines Verzweigungsbefehls finden wir in Abb. 42. Hier muß ein Sprung in Abhängigkeit von einem „ausgeschobenen" Bit erfolgen. Die Kombination von Schiebebefehl LSR und anschließendem Verzweigungsbefehl BCS führt zum gewünschten Ergebnis. LSR schiebt sämtliche Bits des betreffenden Registers eine Stelle nach rechts und der erste Bitwert vor diesem Arbeitsgang wird an das Carry-Bit C weitergegeben. Dieses Carry-Bit hat nun den Wert des „ausgeschobenen" Bits. Wenn C = 1, führt der Verzweigungsbefehl BCS zum Unterprogramm mit dem 2400-Hz-Taktgeber.

10.8 Unbedingte Sprungbefehle (Jump)

In Abb. 16 ist ein Beispiel für einen unbedingten Sprungbefehl angegeben. Nach dem Laden von Register S mit „1" oder „0" für das Aufleuchten der grünen oder roten Signallampe muß auf „Loop" zurückgesprungen werden, so daß das Programm immer wieder neu durchlaufen wird, um den Stand von Weiche und Brücke zu überprüfen. Der Befehl hierzu lautet:

JMP Jump to new location.

Dem JMP-Befehlscode folgt die Adresse zur Fortsetzung des Programms. Es wird angenommen, die Adresse ist 027A und der JMP-Befehlscode steht auf 0300, dann verläuft der Vorgang entsprechend dieser Tabelle:

Adressenbus	*Datenbus*		*Umschreibung*
0300	JMP		Springe nach
0301	7A	—	Adresse
0302	02		027A
027A	Opcode		nächster Befehl

Während auf dem Adressenbus die Adresse 0300 steht, enthält der Datenbus den Operationscode JMP. Der Adressenbus führt die Adresse 0301, wo das niedrigste Byte der neuen Adresse steht. Auf 0302 steht das höchstwertigste Byte. Dementsprechend finden wir anschließend die neue Adresse auf dem Adressenbus und auf dem Datenbus den Operationscode, der auf dieser Adresse gespeichert ist.
Als zweiter Jump-Befehl folgt ein Unterprogrammsprung:

JSR Jump to new location, saving return address (Jump to subroutine)

Der Unterschied zu JMP besteht darin, daß der Inhalt des Befehlszählers beim Verlassen des Hauptprogrammes im Stack gespeichert wird. Gegenüber dem vorigen Jump wird nach Durchlauf der Subroutine stets wieder auf die Adresse zurückgekehrt, wo das Hauptprogramm verlassen wurde. Diese Adresse muß deshalb aufbewahrt werden, um sie spä-

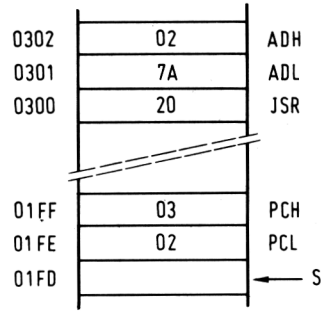


Abb. 50 Typische Speichersituation bei einem JSR-Befehl

ter wieder in den Befehlszähler laden zu können. Nach Durchlauf der Subroutine wird deshalb der Inhalt des Befehlszählers wieder aus dem Stack geholt und auf den Adressenbus gegeben, so daß mit dem Hauptprogramm weiter gearbeitet werden kann, von der Stelle aus, wo dieses verlassen wurde.

Es wird unterstellt, daß der JSR-Opcode ($ 20) auf Adresse 0300 steht. Auf 0301 und 0302 steht dann in der Reihenfolge L–H die Startadresse der Subroutine (z.B.027A).

Die Inhalte der betreffenden Speicherplätze des Programmspeichers und des Stack sind in *Abb. 50* angegeben. Direkt nach dem Decodieren des Operationscodes wird der Stand des Befehlszählers PC (0302) im Stack gespeichert, und zwar in der Reihenfolge H–L (PCH–PCL). Die Abfolge der Operationen ist in untenstehender Tabelle angegeben.

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0300	20 (Opcode)	Fetch Opcode	Beendige vorherg. Befehl, PC wird 0301
2.	0301	7A (neue ADL)	Fetch new ADL	Decodiere ISR, PC wird 0302
3.	01FF	xxx	xxx	Store ADL
4.	01FF	03 (PCH)	Store PCH	Halte ADL S wird 01FE
5.	01FE	02 (PCL)	Store PCL	Halte ADL S wird 01FD
6.	0302	02 (neu ADH)	Fetch ADH	
7.	027A	Opcode	Fetch Opcode	ADL+ 1 → PCL ADH → PCH

Nach dem Opcode-Fetch im ersten Taktzyklus wird im zweiten der Fetch des L-Byte der Startadresse der Subroutine und das Decodieren des Operationscode ausgeführt. Im dritten Zyklus wird die Stack-Adresse 01FF auf den Adreßbus gelegt. Mit dieser Adresse geschieht noch nichts; denn der dritte Zyklus ist erforderlich, um ADL in den Prozessor zu laden und zu speichern. Im vierten Zyklus wird das H-Byte aus dem Befehlszähler in

den Stack, und zwar auf Adresse 01FF (store PCH) und im fünften das L-Byte (PCL) auf Adresse 01FE geladen. Der Inhalt des Befehlszählers geht hierbei nicht verloren. Im sechsten Zyklus kommt der Inhalt des Befehlszählers wieder auf den Adreßbus (0302). Außerdem findet ein Fetch von ADH aus dem Unterprogramm statt. Nunmehr ist die gesamte Adresse der Subroutine im Prozessor und wird im siebenten Zyklus auf den Adressenbus gegeben.
Am Ende der Subroutine finden wir wiederum einen Sprungbefehl:

RTS Return from subroutine.

Dieser Befehl braucht nicht von einer Adresse gefolgt zu werden: Die Adresse, mit der das Hauptprogramm verlassen wurde, steht im Stack. Sie wird aus ihm gelesen und zurück in den Befehlszähler gebracht. Der Vorteil liegt darin, daß eine Subroutine von jeder Stelle in einem Programm angerufen werden kann.
Die Operationsfolge ist nachfolgend angegeben (RTS steht auf Adresse 02A0):

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	02A0	60 (RTS)	Fetch Opcode	beendige vorhergehenden Befehl, PC wird 02A1
2.	02A1	nächster Opcode	ignoriere Opcode	decodiere RTS
3.	01FD	xxx	xxx	S wird 01FE
4.	01FE	02 (PCL)	Fetch PCL	S wird 01FF
5.	01FF	03 (PCH)	Fetch PCH	
6.	0302	02	ignoriere die Daten	PC wird 0303
7.	0303	nächster Opcode	Fetch Opcode	PC wird 0304

Wie aus der Tabelle ersichtlich, kommt nach dem Decodieren des Opcode-RTS während des zweiten Taktzyklus der Inhalt des Stackpointer S auf den Datenbus. Dieser weist noch auf einen leeren Speicherplatz hin, worauf zuerst S im dritten Zyklus auf 01FE angehoben werden muß. Im vierten und fünften Zyklus wird der Inhalt des Befehlszählers PC aus dem Stack geholt, nunmehr in der Reihenfolge L–H, das entspricht der Arbeitsweise des Stack: last in, first out.

Der Befehlszähler wird nun geladen, steht aber noch auf einer Adresse, die zum letzten Befehl des Hauptprogramms gehört, bevor dieses verlassen wurde. Der Befehlszähler muß jedoch zuerst erhöht werden, bevor der Opcode-Fetch ausgeführt werden kann.

Die Möglichkeit besteht, aus einem Programm von einer Subroutine aus eine zweite anzurufen, und vielleicht von hier eine dritte usw. In *Abb. 51* steht im Hauptprogramm auf Adresse 0300 der Befehl JSR, gefolgt von Adresse 027A als Startadresse der ersten

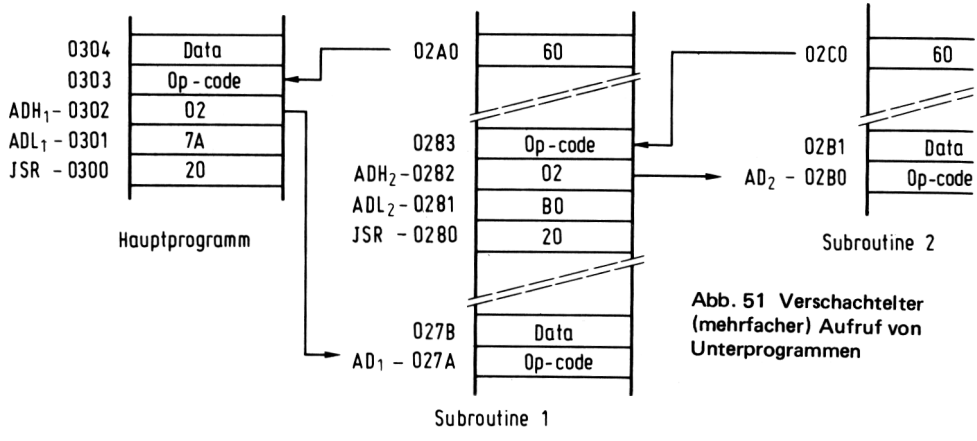


Abb. 51 Verschachtelter (mehrfacher) Aufruf von Unterprogrammen

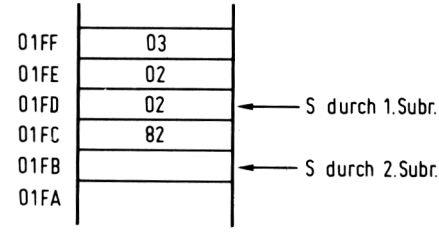


Abb. 52 Stackbelegung bei verschachtelten Unterprogrammen

Subroutine. Die letzte Adresse des Befehlszählers wird in den Stack geladen. Der Stackpointer führt auf 01FD und weist diese Adresse als leeren Speicherplatz an (*Abb. 52*). Die Subroutine 1 startet auf 027A, auf Adresse 0280 dieser Subroutine folgt JSR und anschließend Adresse 02B0. Die letzte Adresse, auf der der Befehlszähler steht, ist 0282 und wird im Stack gespeichert. Der Stackpointer geht nach 01FB. Die zweite Subroutine startet auf 02B0, gefolgt von 02C0 RTS (60). Die Adresse 0282 wird aus dem Stack geholt und der Stackpointer geht auf 01FD. Der Befehlszähler wird um 1 erhöht, und zwar auf 0283, und mit dem Operationscode dieser Adresse wird die erste Subroutine beendet. An ihrem Ende (Adresse 02A0) folgt wieder RTS. Der Befehlszähler wird mit 0302 aus dem Stack geladen, der Stackpointer geht auf 01FF und der Befehlszähler wird auf 0303 erhöht, von wo aus das Hauptprogramm weiter abgearbeitet wird.

Es wird nochmals darauf hingewiesen, daß die Adressen vollkommen beliebig gewählt wurden:
Der Sprung nach einem Interrupt-Programm verläuft ungefähr so wie der Sprung nach einer Subroutine, jedoch rettet die CPU dabei auch das Statusregister auf den Stack. Am Ende eines Interrupt-Programms steht (muß darum stehen) der Befehl:

RTI Return from interrupt.

Es findet nun ein ähnlicher Ablauf statt wie bei RTS. Da bei einem Interrupt nicht nur der Befehlszähler, sondern auch das Statusregister im Kellerspeicher zu speichern sind, werden bei einem RTI selbstverständlich beide Register in die CPU geladen.
Ein Interrupt kann auch durch Software erfolgen. Hierzu ist folgender Befehl erforderlich:

BRK Forced interrupt (Break command).

Kommt während des Programmablaufs dem Prozessor der Opcode eines BRK-Befehls entgegen ($ 00), dann werden die Adressenpuffer mit der Adresse FFFE und ein Taktzyklus später mit FFFF geladen. Auf diese Speicherplätze muß die Startadresse (Vektor) eines Interrupt-Programms in der Reihenfolge L–H geladen werden. Zuerst sind jedoch durch den Prozessor die bei jedem Interrupt notwendigen Schritte vorzunehmen, d.h. Befehlszähler und Statusregister im Stack speichern, sowie Break-Flag B setzen.

Dieses Bit im Statusregister wird nur dann „1", wenn der Prozessor den BRK-Befehl erhalten hat. Der BRK-Befehl kann nicht mit dem Interrupt-Flag I maskiert werden. Wenn I = 1 ist, ruft der Befehl ebenfalls eine Unterbrechung hervor. Den Prozessor-Ablauf bei einer BRK-Instruktion gibt nachstehende Tabelle wieder:

Taktzyklus	*Adressenbus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0300	00 (BRK)	Fetch Opcode	Vollende letzten Befehl. PC wird 0301
2.	0301	Daten	Ignoriere Daten	Decodiere BRK PC wird 0302
3.	01FF	03 (PCH)	Store PCH	S wird 01FE
4.	01FE	02 (PCL)	Store PCL	S wird 01FD
5.	01FD	P	Store P	S wird 01FC
6.	FFFE	7A Vektor L	Fetch Vektor L	
7.	FFFF	02 Vektor H	Fetch Vektor H	7A geht nach PCL
8.	027A	Opcode	Fetch Opcode	027B geht nach PC

In vorstehender Tabelle steht der BRK-Opcode auf 0300. Der Vektor für das Interrupt-Programm ist 027A.

Nun zum „normalen" Hardware-Interrupt. IRQ verläuft in Übereinstimmung mit dieser Tabelle; während der ersten zwei Zyklen wird PC jedoch nicht erhöht. Bei einem IRQ und den gleichen Adressen wie in der vorhergehenden Tabelle muß dann als Inhalt des PC die Adresse 0300 im Stack gespeichert sein.

Das gilt auch für den NMI-Interrupt. In diesem Falle kommen in die Zyklen sechs und sieben die Vektor-Adressen FFFA bzw. FFFB auf die Adressenleitungen. Am Ende eines jeden Interrupt-Programms folgt der Befehl RTI ($ 40), um auf das Hauptprogramm zurückspringen zu können. Die Reihenfolge der Operationen des RTI-Befehls auf Adresse 02A0 ist:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*interner Vorgang*	*externer Vorgang*
1.	02A0	40 (RTI)	Fetch RTI	Letzten Befehl vollenden, PC wird 02A1
2.	02A1	xxx	xxx	Decodiere RTI
3.	01FC	xxx	xxx	S wird 01FD
4.	01FD	P	Fetch P	S wird 01FE
5.	01FE	02 (PCL)	Fetch PCL	S wird 01FF
6.	01FF	03 (PCH)	Fetch PCH	Lade Befehlszähler
7.	0302	op-code	Fetch Op-code	PC wird 0303

Auf Adresse 02A1 sind die Daten für den Befehl unbedeutend. Im dritten Taktzyklus wird der Stapelzeiger auf 01FD erhöht, die erste Adresse aus dem Stack mit dem Inhalt des Statusregisters. Wie beim Stack wird zuerst der Stackpointer erhöht und dann eben ausgelesen. Nach dem Statusregister P wird der Befehlszähler PC geladen, so daß im siebenten Taktzyklus die Adressenleitungen Adresse 0302 angeben.

Aus den Beispielen geht hervor, daß der Befehlszähler nach dem RTI-Befehl zwei Adressen höher steht als die Adresse des BRK-Befehls.

Das ist bei einem Hardware-Interrupt nicht der Fall. Wenn der Prozessor mit einem Befehl beschäftigt ist, wenn ein Interrupt auftritt, dann führt der Prozessor erst den Befehl aus. Der Zählerstand mit der Adresse des nächsten Operationscode wird in den Stack geladen, ebenso der Inhalt des Statusregisters. Das kommt daher, daß bei einem Hardware-interrupt während der ersten zwei Taktzyklen, wie bereits erwähnt, der Befehlszähler nicht erhöht wird. Nach dem RTI-Befehl ist das Befehlsregister in der Lage, den nächsten Opcode, mit dem das Programm weiterlaufen muß, zu lesen.

Sowohl der Software-Interrupt BRK als auch der Hardware-Interrupt IRQ verwenden den gleichen Vektor für das Interrupt-Programm (Vektoren in den Speicherplätzen FFFE und FFFF). Das bedeutet, daß sowohl bei dem Hardware-Interrupt IRQ als auch bei dem Software-Interrupt BRK dasselbe Interrupt-Programm gestartet wird. Es ist trotzdem möglich, daß beide Interrupts verschiedene Programme durchlaufen.

In *Abb. 53* ist der Vektor 0200 auf die Adressen FFFE und FFFF geladen. Das Interrupt-Programm startet auf der Adresse, die der Vektor angibt. Befehlszähler und Statusregister sind im Stack gespeichert. Der Akku wird mit dem Statusregister P durch PLA geladen. Dieses Register ist als letztes im Stack gespeichert. Der Stapelzeiger S kommt auf den Adressenbus 01FD. Dieser muß jedoch nach 01FD, um durch RTI-Befehl die Register wieder in der richtigen Reihenfolge aus dem Kellerspeicher zu holen. Dazu ist Befehl PHA erforderlich. Da auf Adresse 01FD der Inhalt des Statusregisters gespeichert ist (beim Auslesen verändert sich der Inhalt eines Registers nicht), kann der Inhalt durch die PHA-Instruktion nicht geändert werden. Dagegen ändert sich der Inhalt des Stapelzeigers. Dieser zeigt auf den leeren Platz 01FC hin, wie vor dem PLA-Befehl der Adresse 0200.

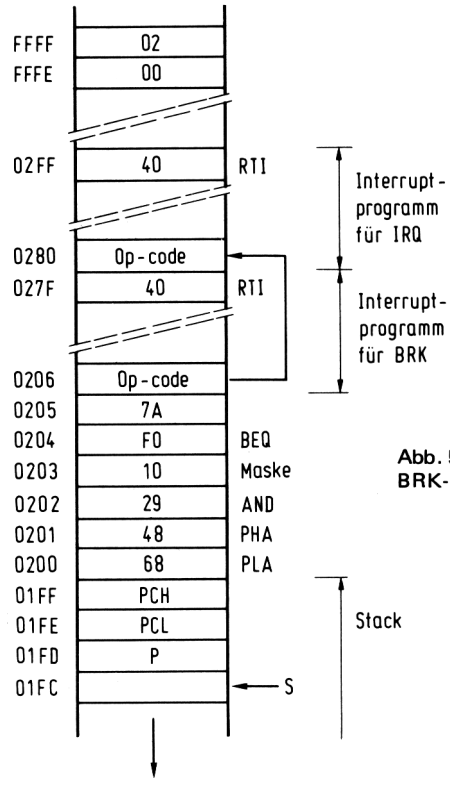


Abb. 53 Wahlweiser Ansprung der BRK- oder IRQ-Routine

Nunmehr folgt ein AND-Befehl mit Maske $ 10. Es wird angenommen, daß der Inhalt des Statusregisters wie folgt ist:

N	V	I	B	D	I	Z	C
1	0	1	0	0	0	0	0

Es findet deshalb ein Hardware-Interrupt durch B = „0“ statt.

Obwohl I im Statusregister gesetzt ist, zeigt sich dies nicht am Inhalt des Kellerspeichers. Das I-Flag wird nämlich im siebenten Taktzyklus als Reaktion auf den Interrupt gesetzt. Der Inhalt des Statusregisters wird im fünften Taktzyklus in den Kellerspeicher gespeichert, so daß B3 der Adresse 01FD „0“ ist. Wäre das nicht der Fall gewesen und Bit 3 wäre 1, dann hätte auf den Interrupt-Request IRQ nicht reagiert werden können. Der AND-Vorgang ist wie folgt:

a	10100000	Statusregister
M	00010000	Maske
a∧M	00000000	Z = 1.

Der BEQ-Befehl auf 0204 erhöht nun den Befehlszähler um 122 Adressen:

$122_{(10)} = 7A_{(16)}$ (Abb. 53).

Wir gelangen dann auf Adresse 0280, von der aus das Interrupt-Programm für den IRQ startet (der Befehlszähler wird erst auf 0206 erhöht und springt dann auf 0280).

Wenn der Interrupt durch eine BRK-Instruktion entstanden ist, so ergibt sich der Inhalt des Statusregisters wie folgt:

N	V	1	B	D	I	Z	C
1	0	1	1	0	0	0	0

Daher: B = 1. Der AND-Vorgang ist nun:

a	10110000	Statusregister
M	00010000	Maske für Break-Flag
a∧M	00010000	Z = 0

Jetzt findet kein Sprung statt. Der Befehlszähler wird von Adresse 0205 auf 0206 erhöht und startet dann das BRK-Interrupt-Programm.

Der Software-Interrupt BRK wird beim „Debugging" (Fehlerbeseitigung) eines Programms benötigt. Es ist kaum anzunehmen, daß ein Programmentwurf sofort fehlerlos arbeitet. Im allgemeinen enthält das Programm Fehler, und die müssen erst aufgespürt werden. Der BRK-Befehl ist hier ein Hilfsmittel. Indem man in einem Programm einen Opcode durch den Opcode vom BRK-Befehl ersetzt, wird das in Frage stehende Programm bis zu dem BRK-Befehl normal durchlaufen. Dann wird auf ein Interrupt-Programm gesprungen, das uns in die Lage versetzt, bestimmte wichtige Register (z.B. die Prozessor-Register) zu überprüfen und festzustellen, ob das Unterprogramm die Erwartungen erfüllt.

Wir benutzen als Beispiel das Programm aus Abb. 3 mit den Daten:

a = 03
b = 05
c = 02

und beschäftigen uns dann mit dem Arbeitsablauf. Wir finden für dieses Programm folgende Befehlsreihe:

CLD
CLC
LDA Adresse a
ADC Adresse b
SBC Adresse c
LSR

Das Ergebnis muß der Zahl im Akku entsprechen:

00000011 C = 0

Bei einem LSR mit einer geraden Zahl ist C „0" und einer ungeraden „1", ebenso wie bei einer geraden Zahl B0 „0" und bei einer ungeraden Zahl B0 „1" ist. Die Antwort lautet deshalb hier:

$$\frac{03 + 05 - 02}{2} = \frac{06}{2} = 03 \quad C = 0$$

Beim Überprüfen der Antwort finden wir jedoch 02 und C = 1.

Es muß deshalb ein Fehler im Programm vorliegen. Nunmehr bringen wir anstelle von SBC den Befehl BRK in das Programm:

```
CLD
CLC
LDA  Adresse a
ADC  Adresse b
BRK
```

Eine Kontrolle nach dem Programmdurchlauf ergibt für den Akkuinhalt:

00001000 C = 0

Dies ist richtig, denn

```
          C          0
          A   00000011
          b   00000101+
          ___________
A+b+C→    A   00001000    C = 0
```

Der Akku enthält die Zahl a und nach dem Arbeitsvorgang das Ergebnis. Der BRK-Befehl wird nun anstelle des LSR-Befehls eingesetzt:

```
CLD
CLC
LDA  Adresse a
ADC  Adresse b
SBC  Adresse c
BRK
```

Nunmehr ist das Ergebnis im Akku:

00000101 C = 1

Dieses Mal ist das Resultat unrichtig, es liegt ein Subtraktionsfehler vor. Es wurde gerechnet:

```
          C          0 ←
          A   00001000 (Ergebnis von ADC)
          c   11111101+
          ___________
A−c−C̄→    A   00000101  C = 1
```

Die Berechnung muß aber lauten:

```
          C          1 ←
          A   00001000
          c   11111101+
          ___________
A−c−C̄     A   00000110  C = 1
```

Dem Befehl SBC muß deshalb der SEC-Befehl vorausgehen. Somit ergibt sich folgendes Programm:

```
CLD
CLC
LDA  Adresse a
ADC  Adresse b
SEC
SBC  Adresse c
LSR
```

10.9 Übrige Befehle

In dieser Gruppe sind die Befehle eingeordnet, die auf das Statusregister Einfluß haben. Hierunter fallen folgende Befehle:

```
CLC    CLI    CLD    CLV
SEC    SEI    SED
```

Nachfolgende Befehle sind bisher noch nicht genannt:

```
CLI  Clear interrupt disable bit.
SEI  Set interrupt disable status.
CLV  Clear overflow flag.
```

Die Befehle sprechen für sich selbst und bedürfen keiner weiteren Erklärung. Der letzte noch nicht genannte Befehl ist

```
NOP  No operation.
```

Besonders bei umfangreichen Programmen ist zu empfehlen, noch einige Stellen frei zu lassen, um, wenn erforderlich, bei einem nicht korrekt arbeitenden Programm einen oder mehrere Befehle einfügen zu können, ohne das gesamte Programm umschreiben zu müssen. Das Programm darf aber durch die Freiplätze nicht unterbrochen werden. Es muß eine fortlaufende Befehlsreihe bestehen bleiben. Diese Freiplätze können aufgefüllt werden, indem man die betreffenden leeren Speicherplätze mit dem Befehl NOP belegt. Das hat keinen Einfluß, sorgt jedoch dafür, daß ein Programm ohne Unterbrechung ablaufen kann. Auch kann dieser Befehl zum Auffüllen von leer gewordenen Speicherplätzen benutzt werden, wenn einige Befehle gestrichen worden sind.

11 Nicht indizierte Adressierungsarten

Die Überschrift dieses Abschnittes besagt gleichzeitig, daß es auch indizierte Adressierungsarten gibt. Der Unterschied zwischen diesen beiden Arten wird erst deutlich nach Abhandlung der letzten Adressierungsart im nächsten Abschnitt. Wir sollten deshalb jetzt von einer Definition absehen.

Der Begriff „Adressierung" kennzeichnet eine Methode zum Auffinden eines Operanden nach einem Operationscode.

Nicht nur der Umfang des Befehlssatzes (Anzahl der verschiedenen Befehle), sondern auch die Anzahl und Art der Adressierungsmethoden bestimmen die Einsatzmöglichkeiten eines Prozessors.

Die CPU 6502 kennt folgende nicht-indizierte Adressierungsarten:

a) unmittelbar (immediate)
b) Zero Page
c) direkt (absolut)
d) indirekt
e) relativ
f) Akku
g) implizit

Indizierte Adressierungsarten sind:

a) Zero Page, x
b) Zero Page, y
c) direkt, x
d) direkt, y
e) indirekt, x (vorindiziert)
f) indirekt, y (nachindiziert)

11.1 Unmittelbare Adressierung

Bei dieser Adressierungsart befindet sich der Operand im Programmspeicher. Operationscode und Operand stehen unmittelbar hintereinander im Programm:

Adresse	*Inhalt*
0200	LDA
0201	$ 04

Der Operand $ 04 folgt direkt auf den Operationscode, der gesamte Befehl besteht daher aus zwei Byte. Im allgemeinen wird diese Adressierungsart für unveränderliche Operanden eingesetzt, z.B. bei einer Maske oder einem Wert, der in einen Zähler eingegeben werden soll. Der Arbeitsgang des Prozessors ist wie folgt:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	Opcode (LDA)	Fetch Opcode	Vollende letzten Befehl. PC wird 0201
2.	0201	Daten (04)	Fetch Data	decodiere Opcode PC wird 0202
3.	0202	Opcode	Fetch Opcode	führe LDA-Befehl aus. PC wird 0203

Der gesamte Befehl läuft innerhalb von zwei Taktzyklen ab. Im dritten wird der decodierte Opcode herausgeführt. Die unmittelbare Adressierung kann folgenden Befehlen zugeordnet werden:

LDA, LDX, LDY, ADC, SBC, AND, ORA, EOR, CMP, CPX, CPY.

11.2 Zero-Page-Adressierung

Es wurde bereits erwähnt, daß in der CPU 6502 im allgemeinen für die Speicherung variabler, häufig benötigter Daten die erste Seite (Seite $ 00) des Speichers eingesetzt wird. Man spricht daher von der Zero-Page-Adressierungsmöglichkeit. Hierbei folgt dem Operationscode des Befehls das niedrigste Byte einer Adresse auf Seite $ 00 mit dem hierin gespeicherten Operanden. Das höchste Byte braucht nicht erwähnt zu werden. Im Hinblick auf das Adressieren sämtlicher anderen Seiten, von denen die vollständige Adresse angegeben werden muß, ermöglicht diese Adressierungsart die Einsparung eines Speicherplatzes pro Befehl.
Beispiel:

Adresse	*Inhalt*
0200	LDA
0201	02

Jetzt muß der Akku mit $ 04 geladen werden. Dieser Operand steht jedoch auf Adresse 0002. Dem LDA-Befehl folgt $ 02, das seine Adresse an Seite $ 00 mit den eingegebenen Daten weitergibt. (*Abb. 54*). Hier der Prozessorvorgang:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	Opcode (LDA)	Fetch Opcode	vollende letzten Befehl PC wird 0201
2.	0201	ADL (02)	Fetch ADL	decodiere LDA PC wird 0202
3.	0002	Daten (04)	Fetch Data	
4.	0202	Opcode	Fetch Opcode	führe LDA-Befehl aus. PC wird 0203

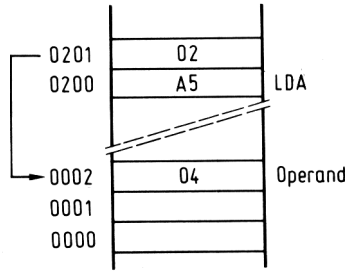


Abb. 54 Zero-Page-Adressierung

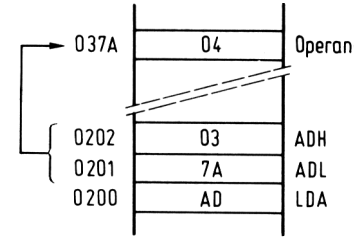


Abb. 55 Direkte (oder absolute) Adressierung

Im dritten Zyklus wird Adresse 0002 auf den Adreßbus gelegt und der Datenblock gelesen. Im vierten Zyklus kommen wir wieder auf das Programm mit Adresse 0202 zurück, der Adresse des nächsten Opcode. Die Zero-Page-Adressierung kann bei folgenden Befehlen angewandt werden:

LDA, LDX, LDY, STA, STX, STY, ADC, SBC, INC, DEC, AND, ORA, EOR, CMP, CPX, CPY, BIT, ASL, LSR, ROL und ROR.

11.3 Direkte Adressierung

Wird ein Datenblock in einen anderen Speicherabschnitt eingegeben als auf Seite $ 00, muß dem Operationscode mit Bezug auf einen Operanden die vollständige Operandenadresse folgen. Für diese Adresse sind zwei Byte erforderlich. Dem Opcode folgt direkt das niedrigste Byte (ADL) und danach das höchste (ADH) der Adresse.
In *Abb. 55* steht der Operand $ 04 auf Adresse 037A, der Opcode für LDA (AD) auf Adresse 0200. Auf 0201 muß somit ADL (7A) folgen und auf 0202 ADH (03). Untenstehende Tabelle vermittelt den Arbeitsvorgang in der CPU:

Taktzyklus	*Adressenbus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	AD (LDA)	Fetch Opcode	Beende vorhergehenden Befehl. PC wird 0201
2.	0201	7A (ADL)	Fetch ADL	decodiere LDA, PC wird 0202
3.	0202	03 (ADH)	Fetch ADH	PC wird 0203
4.	037A	04 (data)	Fetch Data	
5.	0203	neuer Opcode	Fetch Opcode	führe Befehl LDA aus, PC wird 0204

Diese Adressierungsart kann auf folgende Befehle angewendet werden:

LDA, LDX, LDY, STA, STX, STY, ADC, SBC, INC, DEC, AND, EOR, CMP, CPX, CPY, BIT, JMP, JSR, ASL, LSR, ROL und ROR.

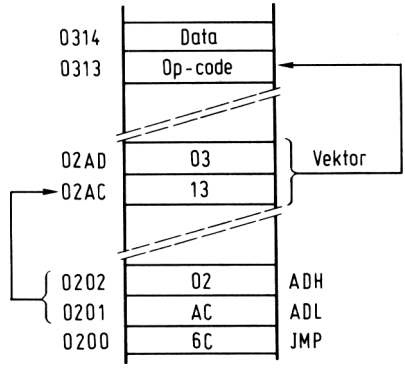


Abb. 56 Indirekte Adressierung über Vektor

11.4 Indirekte Adressierung

Die indirekte Adressierung haben wir im Prinzip bei den Interrupt-Vektoren bereits kennengelernt. Auf zwei Speicherplätzen steht eine Adresse als Anfangsadresse eines Programms. Diese Adressierung für „Hardware"-Operationen kennen wir bereits für IRQ- und NMI-Interrupt, und außerdem auch für den „Reset" (Anschluß 40 der CPU).

Auch der BRK-Befehl gebraucht im Prinzip diese Adressierungsart. Die Vektoren stehen dabei in hierfür fest bestimmten Speicherplätzen. Das ist bei dem JMP-indirekt-Befehl nicht der Fall. Hierbei wird eine Speicherzelle bezeichnet, wo der Vektor der Zieladresse steht, auf die gesprungen werden soll. Zu diesem Zweck folgen dem Opcode das niedrigwertigste und höchstwertigste Byte einer Adresse, wo das niedrigwertigste Byte vom Vektor gefunden wird.

In *Abb. 56* steht der Opcode für den Jump-Befehl (JMP) auf Adresse 0200. Danach folgen die Bytes ADL und ADH, die zusammen die Vektor-Adresse L bilden. Eine Adresse weiter finden wir den Vektor H. Vektor H und Vektor L bilden zusammen die erste Adresse des Teilprogramms, auf das gesprungen werden soll.
Der Arbeitsablauf im Prozessor erfolgt nach untenstehender Tabelle:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	6C (JMP)	Fetch Opcode	Vollende vorhergehenden Befehl. PC wird 0201
2.	0201	AC (ADL)	Fetch ADL	Decodierte JMP. PC wird 0202
3.	0202	02 (ADH)	Fetch ADH	Store ADL
4.	02AC	13 (Vektor L)	Fetch Vector L	ADL wird ADL +1 (AD)
5.	02AD	03 (Vektor H)	Fetch Vector H	Store Vector L
6.	0313	neuer Opcode	Fetch Opcode	PC wird 0314

Diese Adressierungsart ist nur auf den JMP-Befehl anwendbar.

11.5 Relative Adressierung

Die relative Adressierung wird ausschließlich bei Verknüpfungsbefehlen gebraucht. Nach einem Operationscode folgt für eine Verzweigung, wie bekannt, eine sog. Distanzadresse, die angibt, um wieviel Speicherplätze vor- oder zurückgesprungen werden soll. Angenommen, ein Verknüpfungsbefehl auf Adresse 0200 soll um vier Adressen weiter springen, dann muß der Prozessor wie folgt vorgehen:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	Opcode Branch	Fetch Opcode	Vollende vorhergehenden Befehl. PC wird 0201
2.	0201	$ 04	Fetch Daten	Decodiere Verknüpfungs-befehl. PC wird 0202
3.	0202	Opcode	Negiere Opcode	Kontrolliere N-, Z-, V- oder C-Flag, erhöhe PC um $ 04
4.	0206	neuer Opcode	Fetch Opcode	PC wird 0207

Hier fand ein Sprung statt. Fehlt eine derartige Sprunganweisung, muß im dritten Zyklus ein Opcode-Fetch ausgeführt werden. Es muß darauf hingewiesen werden, daß für die Berechnung der neuen effektiven Sprungadresse die Anzahl der Speicherplätze für den Sprung (hier $ 04) zum Inhalt des Befehlszählers (0202) addiert werden muß. Diese Addition ergibt sich im Beispiel deshalb zu: 0202 + 0004 = 0206.

Im allgemeinen ist die Distanz, d.h. um wieviel Speicherplätze der Befehlszähler übersprungen werden muß (die Sprunggröße), nicht bekannt, dagegen aber die Adresse, wohin gesprungen werden muß. Die Berechnung des Vorwärtssprungs verläuft dann so: Angenommen, der Operationscode des Verknüpfungsbefehls befindet sich auf der Adresse 02BA und es soll nach der Adresse 030C gesprungen werden. Zuerst ist also die Opcode-Adresse um zwei auf 02BC zu inkrementieren. Nun wird gerechnet:

	hex	*binär*	
Sprungadresse	030C	00000011	00001100
Opcode-Adresse +2	02BC–	00000010	10111100–
	0050	00000000	01010000

Der vollständige Befehl lautet nunmehr:

Opcode
$ 50

Auch bei einem Rückwärtssprung muß die Opcode-Adresse um zwei inkrementiert werden. Wir nehmen an, die Opcode-Adresse ist 0325- und die Sprungadresse 02D4, dann gilt:

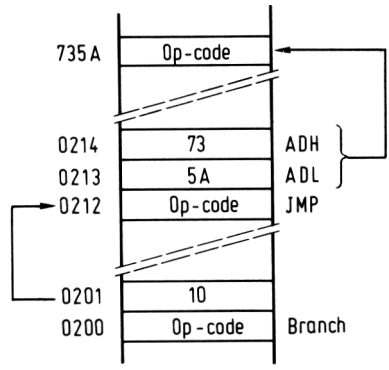


Abb. 57 Relative Adressierung durch Versatzadresse

muß zuerst die Opcode-Adresse um zwei inkrementiert werden:
0325+0002=0327

	hex	*binär*	
Sprungadresse	02D4	00000010	11010100
Opcode Adresse +2	0327–	00000011	00100111–
	FFAD	11111111	10101101

Der Befehl lautet nunmehr:

Opcode
$ AD

Die Distanzgröße für einen Vorwärtssprung ist positiv ($ 01 bis $ 7F), für einen Rückwärtssprung negativ ($ FF bis 80). Die Sprünge können deshalb nicht weiter als 127 Speicherplätze sein. Ist der Sprung größer, dann muß nach einer Zwischenadresse gesprungen werden. Auf dieser Adresse befindet sich dann ein JMP-Befehl zur endgültigen Zieladresse. In *Abb. 57* steht der Verknüpfungsbefehl auf 0200. Die Sprungadresse ist 735A. Da dieser Sprung für eine Verzweigung zu groß ist, sorgt ein JMP-Befehl auf 0212 für eine Zwischenstufe.

11.6 Akku und implizite Adressierung

Bei Akku und impliziter Adressierung beziehen sich die Befehle auf die CPU-Register und bedürfen daher keiner weiteren Adressierung. Es sind ein-Byte-Befehle, die angewandt werden bei:

ASL, LSR, ROL, ROR, für den Akku und weiterhin
TAX, TAY, TXA, TYA, TXS, TSX, PLA, PHA, PLP, INX, DEX, INY, DEY, CLC, CLD, CLI, CLV, SEC, SED, SEI, NOP, RTS, RTI und BRK.

Obwohl der BRK-Befehl ein Ein-Byte-Befehl ist und insofern unter der Bezeichnung implizite Adressierung läuft, gleicht die Arbeitsweise im Prinzip einer indirekten Adressierung. Die Arbeitsvorgänge im Prozessor für einen TAX-Befehl lauten:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	Opcode (TAX)	Fetch Opcode	vollende vorhergehenden Befehl, PC wird 0201
2.	0201	Opcode	Ignoriere Opcode	decodiere TAX
3.	0201	Opcode	Fetch Opcode	führe Befehl TAX aus, PC wird 0201

12 Indizierte Adressierungsarten

12.1 Zero-Page-indizierte Adressierung

Bei den indizierten Adressierungsarten wird der Inhalt eines Index-x- oder Index-y-Registers benötigt. Dabei wird immer der Inhalt eines Index-Registers zu einer gegebenen Relativ-Adresse addiert, um die endgültige Operanden-Adresse zu erhalten.
Der Nutzen einer indizierten Adressierung geht aus *Abb. 58* hervor. Die ersten sechs Speicherplätze auf Seite $ 00 müssen mit $ 00 geladen werden. Als Zähler dient das Index-x-Register. Der Inhalt dieses Registers bestimmt gleichfalls die Adresse, die eingeladen werden soll. Als Basis-Adresse ist $ 0000 angegeben. Die endgültige Adresse ergibt sich dann zu 0000+x.

Die erste einzuspeichernde Adresse ist dann

0000 + 05 = 0005

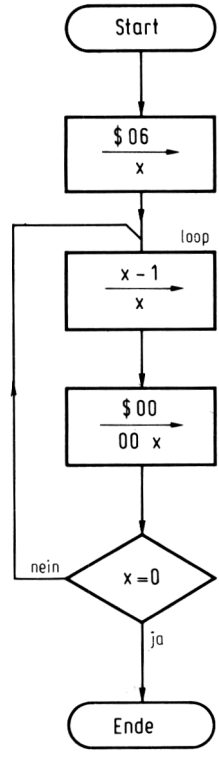


Abb. 58 Indexregister als Zähler in einer Schleife

Da der Zähler in jedem Zyklus um 1 dekrementiert wird, ist anschließend der Speicherplatz mit 0004, im nächsten Zyklus mit 0003 usw. bis zum letzten Speicherplatz mit 0000 zu laden.

Der Prozessor-Ablauf bei Zero-Page-Adressierung ergibt sich somit wie folgt (Opcode auf 0205, x = 05):

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0205	95 (STA)	Fetch Opcode	Vollende vorhergehenden Befehl. PC wird 0206
2.	0206	00 (BAL)	Fetch Basis-Adresse L	Decodiere LDA, PC wird 0207
3.	0000 (00, BAL)	xxx	xxx	ADL wird BAL + x = 00 + 05 = 05
4.	0005	Akku (00)	Store Akku	
5.	0207	Opcode	Fetch Opcode	PC wird 0208

Auf diese Weise läßt sich auf Seite $ 01 kein Speicherplatz adressieren. Angenommen, die Basisadresse ist 00E5 und der Inhalt vom Index-x-Register $ 20, dann sollte entsprechend der Addition 00E5 + 20 = 0105, der Speicherplatz 0105 adressiert werden.

Der adressierte Speicherplatz wird aber 0005, folglich eine Adresse auf Seite $ 00. Angenommen, das Programm nach Abb. 58 startet auf Adresse 0200, dann ergibt sich eine Befehlsfolge nachfolgender Tabelle:

Adresse	*Kennung*	*Mnemonic-Symbol*	*Adressierungsart*	*Speicherinhalt*
0200		LDA	immediate	A9
0201		$ 00		00
0202		LDX	immediate	A2
0203		$ 06		06
0204	loop	DEX	implied	CA
0205		STA	zero page, x	95
0206		$ 00		00
0207		BND	relative	D0
0208		FB		FB
0209				

Der letzte Arbeitsgang vor dem Sprungauftrag mit Einfluß auf das Statusregister ist DEX. Es muß nach „loop" gesprungen werden, da der Inhalt des Index-x-Registers nicht Null ist, folglich Z = 0. Hierzu dient der BNE-Befehl. Der STA-Befehl zwischen DEX und BNE beeinflußt das Statusregister nicht.

Die Sprungberechnung verläuft folgendermaßen:

	hex	*binär*
Sprungadresse (loop)	04	00000100
Adresse BNE +2	09–	00001001–
	FB	11111011

Der Speicherplatzinhalt 0208 wird FB.
Folgenden Befehlen kann die Zero-Page-x-Adressierung zugeordnet werden:

LDA, LDY, STA, STY, ADC, SBC, INC, DEC, AND, ORA, EOR, CMP, ASL, LSR, ROL und ROR.

In gleicher Weise kann auch das Index-y-Register für die Zero-Page-Indizierung, jedoch nur für die Befehle LDX und STX eingesetzt werden.

12.2 Direkt indizierte Adressierung

Bei der direkt indizierten Adressierung (engl. absolute indexed) besteht die Basisadresse wie bei der direkten Adressierung aus zwei Byte. Diese Adressierungsart wird beim Adressieren auf alle Seiten angewendet. Im Gegensatz zur Zero-Page-Indizierung kann man hier durch Indizierung auch auf eine Adresse in der nächst-höheren Seite als die der Basisadresse gelangen. Ist die Basisadresse 02E5 und der Inhalt des diesbezügl. Indexregisters $ 20, dann ist die angegebene Adresse tatsächlich

02E5 + 20 = 0305.

Liegen Basisadresse und endgültige Adresse auf der gleichen Seite, dann führt der Prozessor nachfolgende Operationen aus (LDA auf Adresse 0200, Basisadresse 0310, x = $ 05):

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	BD (LDA)	Fetch Opcode	Vollende vorhergehenden Befehl, PC wird 0201
2.	0201	10 (BAL)	Fetch BAL	Decodiere LDA, PC wird 0202
3.	0202	03 (BAH)	Fetch BAH	ADL wird BAL + x = 10+05 = 15 PC wird 0203
4.	0315	Daten	Fetch Data	
5.	0203	nächster Opcode	Fetch Opcode	Vollende LDA, PC wird 0204

Liegt die endgültige Adresse eine Seite höher als die Basisadresse, dann ist ein besonderer Taktzyklus erforderlich, um nach der Addition von BAL + x noch das Carry-Bit (1) zu BAH zu addieren, woraus sich dann ADH = BAH + 1 ergibt. Diese Adressierungsart kann für das Index-x-Register mit:

ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC und STA

und das Index-y-Register für:

ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC und STA

angewendet werden.

12.3 Indiziert-indirekte Adressierung mit x

Aus der Überschrift geht hervor, daß es sich hier um eine indiziert-indirekte Adressierungsart handelt, und zwar mit Hilfe des Index-x-Registers auf folgende Weise: Nach dem Operationscode folgt eine Adresse auf Seite $ 00 (Zero-Page-Adressierung). Das ist die Basisadresse. Hierzu wird der Inhalt des Index-Registers addiert. Daraus ergibt sich die indirekte Adresse, in der der Operand gespeichert ist, als ADL und ADH.
In *Abb. 59* finden wir auf Adresse 0200 den Operationscode für LDA mit nachfolgender $ 02 als „L"-Basisadresse (BAL). Hierzu wird der Inhalt des Index-x-Registers (hier 05) addiert, woraus sich

02 + 05 = 07

ergibt.

Das ist das L-Byte der indirekten Adresse auf Seite $ 00 (IAL_1). Auf dieser Adresse steht nun das „L"-Byte der Operanden-Adresse 1 (ADL1). Das „H"-Byte (ADH1) befindet sich im Speicherplatz 0008. ADH_1 und ADL_1 bilden zusammen die Operandenadresse 1 (03A0).

Um die Adresse des nächsten Operanden (2) zu finden, muß der Inhalt des Index-x-Registers um 2 inkrementiert werden, in diesem Fall

05 + 02 = 07.

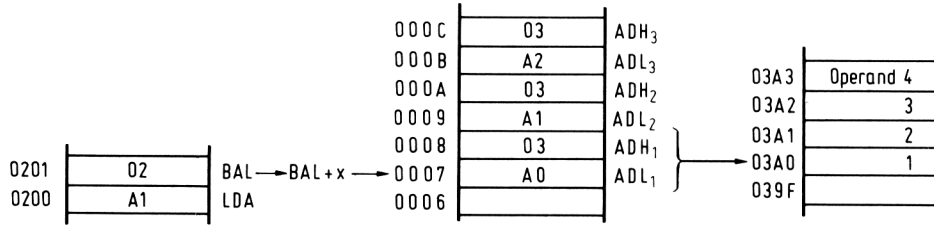


Abb. 59 Wirkungsweise der indiziert-indirekten Adressierung

Das „L"-Byte der indirekten Adresse ist dann:

02 + 07 = 09

Wir finden auf der indirekten Adresse 0009 ADL_2 und auf 000A ADH_2. ADH_2 und ADL_2 bilden zusammen die Operandenadresse 2.

Der Speicherplatzbedarf ist bei dieser Operation beträchtlich. Die indirekten Adressen auf Seite $ 00 erfordern 10 verschiedene Operanden auf 20 Speicherplätzen. Die Operanden können sich an jeder beliebigen Stelle im Speicher befinden.

Bei dieser Adressierungsart, die auch mit indizierter Indirekt-Adressierung bezeichnet wird, muß der Prozessor nachfolgende Operationen ausführen (x = $ 05):

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	A1 (LDA)	Fetch Opcode	Vollende vorhergehenden Befehl. PC wird 0201
2.	0201	02 (BAL)	Fetch BAL	Decodiere LDA. PC wird 0202
3.	0002 (00,BAL)	xxx	xxx	IAL_1 wird BAL + x_1
4.	0007 (00,IAL_1)	A0 (ADL_1)	Fetch ADL_1	Inkrementiere IAL_1 um 1
5.	0008	03 (ADH_1)	Fetch ADH_1	Halte ADL_1
6.	03A0	Operand 1	Fetch Operand 1	
7.	0202	Opcode	Fetch Opcode	Führe Befehl LDA aus, PC wird 0203

Für die Addition von x zu BAL ist ein besonderer Taktzyklus (Zyklus 3) erforderlich.

Diese Adressierungsart ist bei folgenden Befehlen anwendbar:

ADC, AND, CMP, EOR, LDA, ORA, SBC und STA.

12.4 Indirekte-indizierte Adressierung

Diese Adressierungsart weicht in gewisser Weise von der (indirekten)x-Adressierung ab, und zwar nicht nur wegen der Verwendung des Index-y-Registers. Nach dem Opcode folgt direkt das niedrigste Byte der indirekten Adresse auf Seite $ 00. Hier liegt die Basisadresse Low (BAL), im nächsten Speicherplatz High (BAH). Somit sind Basisadresse und durch Addition von BAL mit dem Inhalt des Index-y-Registers auch die Operandenadresse ermittelt (ADH = BAH, ADL = BAL + y).

In *Abb. 60* befindet sich auf Adresse 0200 der Opcode des LDA-Befehls, gefolgt von 07 auf 0201. Letztere ist das L-Byte der indirekten Adresse auf Seite $ 00 (IAL). Auf die-

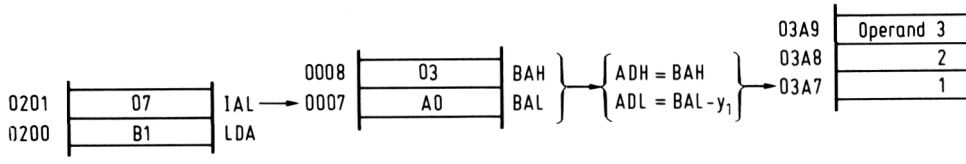


Abb. 60 Wirkungsweise der indirekt-indizierten Adressierung

ser indirekten Adresse wiederum steht das L-Byte der Basisadresse (BAL). Die nächste Adresse beinhaltet das H-Byte der Basisadresse (BAH auf 0008). Durch Inkrementierung von BAL mit dem Inhalt des Index-y-Registers, hier $ 07, wird die Adresse des Operanden gefunden (ADH_1 = BAH, ADL_1 = BAL + y).

Um den zweiten Operanden zu finden, ist das Index-Register I um 1 zu inkrementieren. Um sämtliche Operanden zu adressieren, sind bei dieser Adressierungsart nur zwei Speicherplätze auf Seite $ 00 erforderlich. Die Operanden können jedoch nicht weiter als 256 Speicherplätze (max. Inhalt vom Index-y-Register) auseinander liegen. Unter Zugrundelegung dieser Gegebenheiten führt, wie in Abb. 60, der Prozessor folgende Operationen aus:

Takt-zyklus	*Adressen-bus*	*Datenbus*	*externer Vorgang*	*interner Vorgang*
1.	0200	B1 (LDA)	Fetch Opcode	Vollende vorhergehenden Befehl. PC wird 0201
2.	0201	07 (IAL)	Fetch IAL	Decodiere LDA. PC wird 0202
3.	0007 (00 IAL)	A0 (BAL)	Fetch BAL	Inkrementiere IAL um 1
4.	0008	03 (BAH)	Fetch BAH	ADL wird BAL + y
5.	03A7 (ADH, ADL)	Daten	Fetch Operand	
6.	0202	Opcode	Fetch Opcode	Führe Befehl LDA aus. PC wird 0203

Auf Speicherplätzen zweier aufeinanderfolgender Seiten können Operanden gespeichert werden. In diesem Falle gibt BAH eine zu tiefe Seitennummer an. Jetzt ist noch ein besonderer Taktzyklus erforderlich, um nach der Rechnung ADL = BAL + y – das H-Byte der Adresse aus ADH = BAH + 1 zu ermitteln.

Hier folgen Befehle, denen diese Adressierungsart zugeordnet werden kann:

ADC, AND, CMP, EOR, LDA, ORA, SBC und STA.

13 Einfache Programme

13.1 Allgemeines

Im Rahmen dieses Buches kann kein theoretischer Programmierkursus gegeben werden. Das Programmieren ist nämlich nur zu lernen, wenn ein Computer zur Verfügung steht, um das einwandfreie Arbeiten eines Programms zu kontrollieren und um Fehler aufzufinden. Besonders das Aufspüren von Fehlern und das Verbessern ist zum Erlangen eines guten Einblicks in die Arbeitsweise eines Computers von besonderer Bedeutung.

Wenn zum Einschreiben von Daten ein Tastenfeld und ein Display (evtl. ein Sichtgerät) zum Auslesen benutzt wird, dann muß ein „Monitorprogramm" fest im System vorhanden sein. Es ist leider für jeden Computer verschieden. Hier kann deshalb auch nicht weiter gegangen werden, als daß einige Anleitungen und einige einfache Programme gegeben werden, die als Teilprogramme benutzt werden und als Beispiele dienen können, wie Probleme als Teil eines größeren Programmes zu lösen sind.

Es wurde bereits erwähnt, daß für einen Programmentwurf ein vollständiger Einblick in das Problem, wofür das Programm erstellt werden soll, unbedingt notwendig ist. So muß derjenige, der für ein wissenschaftliches Problem ein Programm entwerfen soll, auch selbst in der Lage sein, dieses Problem zu lösen. Ein Computerprogramm für die Steuerung von Ampeln an einer Kreuzung kann nicht erstellt werden, wenn der Programmierer die möglicherweise auftretenden Verkehrssituationen nicht kennt.

Für ein bestimmtes Problem sind natürlich verschiedene Lösungen möglich. Es liegt am Programmierer, die Lösung herauszusuchen, die für den Computer am besten geeignet ist. Deshalb ist er auch von der Beschreibung und der Gebrauchsanweisung, die der Hersteller zu dem betreffenden Computer liefert, abhängig. Damit soll jedoch nicht gesagt sein, daß für ein Problem ein bestimmter Computertyp erforderlich ist, sondern nur, daß ein Programm spezifisch für einen bestimmten Computertyp geschrieben werden muß.

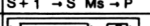
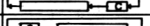
Wenn das Problem analysiert wurde, kann ein Flußdiagramm hergestellt werden. Es ist zu empfehlen, daß dieses Programm nur in groben Zügen aufgestellt werden sollte und daß erst später die einzelnen Blöcke auszuarbeiten sind. So geht die Übersicht über das Programm nicht verloren. Evtl. kann das Ausarbeiten stufenweise bis in immer kleinere Details erfolgen. Das endgültige Flußdiagramm eines Teilprogramms muß so weit ausgearbeitet sein, daß ohne große Schwierigkeiten das Programm in Maschinensprache geschrieben werden kann. (Es gibt auch andere Sprachen, aber hierüber später). Der Begriff „Maschinensprache" läßt sich einfach beschreiben: Es ist die Sprache, die die CPU direkt versteht. Die einzige Sprache, die der Mikrocomputer ohne weitere Hilfsmittel versteht, ist die der Kombination von acht „Einsen" und „Nullen", d.h. binäre Zahlen mit acht Bit. Diese Kombinationen lassen sich leicht in ihrem hexadezimalen Wert ausdrücken. Das bedeutet, daß jede Operation in Codeform eingegeben werden muß (Operationscode).

Die Gesamtanzahl möglicher Operationscodes bei einem 8-Bit-Mikroprozessor beträgt $2^8 = 256$. Das ist mehr als ausreichend. Die CPU 6502 benötigt hiervon nur 146.

Tabelle 1.

Instructions		Immediate			Absolute			Zero Page			Accum.			Implied						(Ind.) Y			Z. Page. X			Abs. X			Abs. X			Relative			Indirect			Z. Page Y			Condition Codes					
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3											✓	✓	✓	–	–	✓
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3											✓	✓	–	–	–	–
ASL	C←[ ]←Ø				ØE	6	3	Ø6	5	2	ØA	2	1										16	6	2	1E	7	3													✓	✓	✓	–	–	–
BCC	Branch on C = Ø (2)																															9Ø	2	2							–	–	–	–	–	–
BCS	Branch on C = 1 (2)																															BØ	2	2							–	–	–	–	–	–
BEQ	Branch on Z = 1 (2)																															FØ	2	2							–	–	–	–	–	–
BIT	A∧M				2C	4	3	24	3	2																															M7	✓	–	–	–	M6
BMI	Branch on N = 1 (2)																															3Ø	2	2							–	–	–	–	–	–
BNE	Branch on Z = Ø (2)																															DØ	2	2							–	–	–	–	–	–
BPL	Branch on N = Ø (2)																															1Ø	2	2							–	–	–	–	–	–
BRK														ØØ	7	1																									B = 1					
BVC	Branch on V = Ø (2)																															5Ø	2	2							–	–	–	–	–	–
BVS	Branch on V = 1 (2)																															7Ø	2	2							–	–	–	–	–	–
CLC	Ø → C													18	2	1																									–	–	Ø	–	–	–
CLD	Ø → D													D8	2	1																									–	–	–	–	Ø	–
CLI	Ø → 1													58	2	1																									–	–	–	Ø	–	–
CLV	Ø → V													B8	2	1																									–	–	–	–	–	Ø
CMP	A–M (1)	C9	2	2	CD	4	3	C5	3	2							01	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3											✓	✓	✓	–	–	–
CPX	X–M	EØ	2	2	EC	4	3	E4	3	2																															✓	✓	✓	–	–	–
CPY	Y–M	CØ	2	2	CC	4	3	C4	3	2																															✓	✓	✓	–	–	–
DEC	M-1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓	✓	–	–	–	–
DEX	X–1 → X													CA	2	1																									✓	✓	–	–	–	–
DEY	Y–1→ Y													88	2	1																									✓	✓	–	–	–	–
EOR	AvM → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3											✓	✓	✓	–	–	–
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓	✓	–	–	–	–
INX	X + 1 → X													E8	2	1																									✓	✓	–	–	–	–
INY	Y + 1 → Y													C8	2	1																									✓	✓	–	–	–	–
JMP	Jump to new Loc.				4C	3	3																													6C	5	3				–	–	–	–	–	–
JSR	(See fig. 2) Jump Sub				2Ø	6	3																																			–	–	–	–	–	–
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3											✓	✓	–	–	–	–
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3									B6	4	2	✓	✓	–	–	–	–
LDY	M → Y (1)	AØ	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													✓	✓	–	–	–	–
LSR	Ø → [ ]→C				4E	6	3	46	5	2	4A	2											56	6	2	5E	7	3													Ø	✓	✓	–	–	–
NOP	No Operation													EA	2	1																									–	–	–	–	–	–

Tabelle 1 (Fortsetzung)

Instructions		Immediate			Absolute			Zero Page			Accum.			Implied			(Ind.) X			(Ind.) Y			Z. Page. X			Abs. X			Abs. X			Relative			Indirect			Z. Page Y			Condition Codes					
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
ORA	AVM + A	Ø9	2	2	ØD	4	3	Ø5	3	2							Ø1	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										√	√	–	–	–	–
PHA	A → Ms S–1 → S													48	3	1																								–	–	–	–	–	–
PHP	P → Ms S–1 → S													Ø8	3	1																								–	–	–	–	–	–
PLA	S + 1 → S Ms → A													68	4	1																								√	√	–	–	–	–
PLP	S + 1 → S Ms → P													28	4	1																								*(Restored)					
ROL						6	3	26	5	2	2A	2	1										36	6	2	3E	7	3												√	√	√	–	–	–
ROR					6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3												√	√	√	–	–	–
RTI	(See fig. 1) RTRN (See fig. 1) INT													4Ø	4	1																								*(Restored)					
RTS	(See fig. 2) RTRN. SUB.													6Ø	6	1																								–	–	–	–	–	–
SBC	A–M–$\overline{C}$ – A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										√	√	(3)	:	:	√
SEC	1 → C													38	2	1																								–	–	1	–	–	–
SED	1 → D													F8	2	1																								–	–	–	–	1	–
SEI	I → I													78	2	1																								–	–	–	1	–	–
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										–	–	–	–	–	–
STX	X → M				8E	4	3	86	3	2																													96	4	2	–	–	–	–	–	–
STY	Y → M				8C	4	3	84	3	2													94	4	2																–	–	–	–	–	–
TAX	A → X													AA	2	1																								√	√	–	–	–	–
TAY	A → Ÿ													A8	2	1																								√	√	–	–	–	–
TSX	S → X													BA	2	1																								√	√	–	–	–	–
TXA	X → A													8A	2	1																								√	√	–	–	–	–
TXS	X → S													9A	2	1																								–	–	–	–	–	–
TYA	Y → A													98	2	1																								√	√	–	–	–	–

(1) Add 1 To "N" If Page Boundry Is Crossed
(2) Add 1 To "N" If Branch Occurs To Same Page
(2) Add 2 To "N" If Branch Occurs to Different Page
(3) Carry not = Borrow
(4) If In Decimal Mode Z Flag Is Invalid:
Accumulator Must Be Checked For Zero Result

Übernommen aus Befehlssatz MOS-Technology, Inc.

X Index X
Y Index Y
A Accumulator
M Memory per effective Address
Ms Memory per Stack Pointer
\+ Add
– Subtract
Λ And
V Or
⊻ Exclusive or
√ Modified
– Not Modified
M7 Memory Bit 7
M6 Memory Bit 6
N No. Cycles
No. Bytes

Das besagt nicht, daß die CPU 146 verschiedene Operationen erkennt. Eine Reihe von Operationen (z.B. LDA, STA usw.) erfordert mehrere Adressierungsarten und für jede Adressierungsart hat ein Befehl einen spezifischen Operationscode. Die Anzahl verschiedener Befehle bei der CPU 6502 beträgt 56 und die der Adressierungsarten 13.

All diese Operationen mit den dazu gehörenden Operationscodes sind in *Tabelle 1* aufgelistet. Hierin ist für jede Adressierungsart ein dreispaltiger Block mit folgender Bedeutung angegeben:

„op" Operationscode für den Befehl,
„N" Zahl der Taktzyklen, die für die Ausführung eines vollständigen Befehls erforderlich sind
„#" Anzahl der Byte für einen Befehl.

Bei Computern mit beschränktem Speicherumfang ist es von Bedeutung, eine Lösungsmethode mit nicht zu großer Byte-Anzahl zu wählen, damit das Programm noch im Speicher Platz findet. Bei wieder anderen Programmen oder Unterprogrammen kommt es auf die Arbeitsgeschwindigkeit an. Diese kann durch Addition der benötigten Anzahl von Taktzyklen pro Befehl und Multiplikation mit der Impulsfolgezeit des Taktsignals (bei einem Taktsignal von 1 MHz beträgt die Impulsfolgezeit 1 μs) ermittelt werden. Es ist zweckmäßig, ein Programm so übersichtlich wie möglich zu schreiben, so daß später der Zweck der Programmschritte einfach zu erkennen ist. Das ist auch besonders für die Fehlersuche von Bedeutung.

In *Tabelle 2* ist das Programm für das in Abb. 58 dargestellte Flußdiagramm angegeben. Es betrifft das Ladeprogramm der ersten sechs Speicherplätze auf Seite $ 00 mit $ 00. Jede Zeile der Tabelle enthält einen vollständigen Befehl, bestehend aus Operationscode, Operanden oder Operandenadresse. In der Spalte „Adresse" (Adr.) steht die erste Befehlsadresse, in Spalte „Code" steht zuerst der Opcode, dann Operand oder Operandenadresse in Abhängigkeit von der Adressierungsart. In Spalte „Label" ist die Bezeichnung für den Befehl angegeben. Sie sollte leicht zu merken sein und bezieht sich im allgemeinen auf einen Sprungbefehl.

Die Spalte „Befehl" (Instr.) enthält das Mnemonik-Symbol für den Befehl und Spalte „Operand" die Bezeichnung für den Operanden, auf den sich die Operation bezieht. Finden wir auf der Zeile der Adresse 0207 die Bezeichnung „loop", so bedeutet das, daß nach der Adresse mit der Bezeichnung „loop" gesprungen werden muß. Die letzte Spalte ist für Bemerkungen zu den Befehlen vorgesehen, um den Programmablauf besser zu verfolgen und den Zusammenhang mit dem Flußdiagramm erkennen zu können.

Tabelle 2. Programm: Laden von Speicherplätzen

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A900	Start	LDA	A	$ 00→A
0202	A206		LDX	x	$ 06→x
0204	CA	loop	DEX	x	5 Schleifen
0205	9500		STA		Z-page x $ 00→00, x
0207	D0FB		BNE	loop	rel. Sprung, wenn x ≠ 0
0209					

Tabelle 3. Programm: Einfache Berechnung

0200	D8	Start	CLD		D=0
0201	18		CLC		C=0
0202	A500		LDA	a	Z-page. a→A
0204	6501		ADC	b	Z-page. a+b
0206	38		SEC		C=1
0207	E502		SBC	c	Z-page. a+b−c
0209	4A		LSR		Akku $\frac{a+b-c}{2}$
020A	8503		STA	M	Z-page. A→M
020C					

Von diesem Programm ist in Verbindung mit der Abb. 58 die Befehlsfolge mit ausführlicher Beschreibung angegeben. Sie kann als Programm für das Eingeben einer Zahl in eine Reihe von Speicherplätzen angesehen werden.

13.2 Einfache Berechnung

Tabelle 3 ist ein Beispiel für eine Reihe von Arbeitsvorgängen, die nacheinander ausgeführt werden müssen:

$$p = \frac{a+b-c}{2}$$

Es handelt sich hier um die Zahlen a, b und c, die nicht größer sein dürfen als 128 (pro Zahl).

Folgende Speicherplätze werden für die drei Zahlen benötigt:

a 0000
b 0001
c 0002
M 0003

Für die Ausarbeitung ist immer Akku A erforderlich. Die ALU übernimmt die Bearbeitung zwischen der Zahl im Akku und einer Zahl in einem Speicherplatz, die in das Hilfsregister geladen wird (Abb. 32). Das Laden der Zahl in den Akku (a → A) übernimmt der LDA-Befehl. Das Laden des Hilfsregisters erfolgt ohne Befehl.

In *Abb. 61* ist das Flußdiagramm von diesem Programm angegeben. Zuerst wird die Zahl a in den Akku geladen: a → A. Dann wird addiert: A + b → A. Dieses Ergebnis ist nun im Akku. Jetzt folgt die Subtraktion A − c → A. Das Ergebnis ist a + b − c und erscheint wiederum im Akku. Zuletzt ist die Division durch 2 durchzuführen; dazu wird die Zahl im Akku um ein Bit nach rechts geschoben und Bit 7 mit Null geladen. Am Programmanfang muß für das Binärrechnen zuerst der Dezimalmodus auf „0", vor der Addition C auf „0" und vor der Subtraktion C auf „1" gesetzt werden. Nach der Rechnung

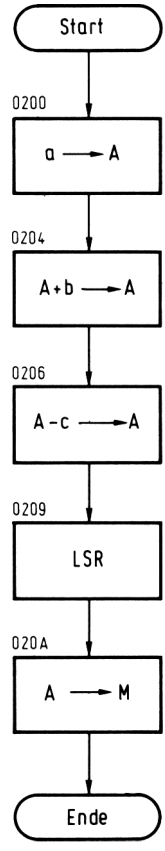


Abb. 61 Flußdiagramm zur Lösung einer arithmetischen Aufgabe

wird das Ergebnis im Speicherplatz 0003 gespeichert (A → M). Das Programm ist in Tabelle 3 angegeben.

Im Flußdiagramm (Abb. 61) sind die Blöcke mit der Adresse des zugehörigen Befehls versehen. Das Verfolgen des Programms wird dadurch vereinfacht.

13.3 Logische Funktionen

Logische Funktionen beziehen sich bei einem Mikrocomputer auf den Inhalt eines Registers. Dieses kann sowohl ein Eingangstor als auch ein Speicherplatz sein. Es muß dann über „richtig" oder „falsch" einer Aussage bezügl. des Registerinhaltes entschieden werden. Die Aussage braucht sich nicht auf den gesamten Registerinhalt zu beziehen, sondern kann auch im Zusammenhang mit dem Zustand eines Bit in diesem Register stehen. In diesem Fall betrifft die Aussage lediglich den Zustand, ob ein Bit auf Null liegt oder nicht.

Bezieht sich die Aussage auf den gesamten Registerinhalt, dann handelt es sich hier nicht um „Null" des Registerinhaltes, sondern um die Beurteilung, ob der Registerinhalt gleich, größer oder kleiner zu einer bestimmten Zahl oder einem anderen Speicherinhalt ist.

Das Programm aus Abb. 8 für die EN-Funktion ist in *Abb. 62* weiter ausgearbeitet. Die Aussagen beziehen sich auf Zahl a auf Adresse 0000 und Zahl b auf Adresse 0001. Das Ergebnis erscheint in M auf Adresse 0002. Ist Zahl a nicht Null, muß nach AND 1 gesprungen werden. Die Aussage a = 0 ist dann falsch (eine „Null" in der Wahrheitstabelle).

Der Sprungbefehl hierfür muß BNE lauten. Dasselbe gilt für die Aussage über die Zahl b. In *Tabelle 4* sind die Befehle aufgeführt:

a = 0	*b = 0*	*M*
0	0	0
0	1	0
1	0	0
1	1	1

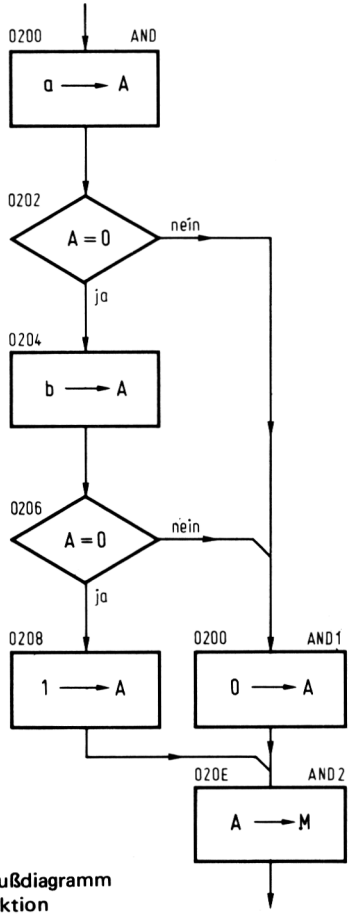


Abb. 62 Flußdiagramm zur EN-Funktion

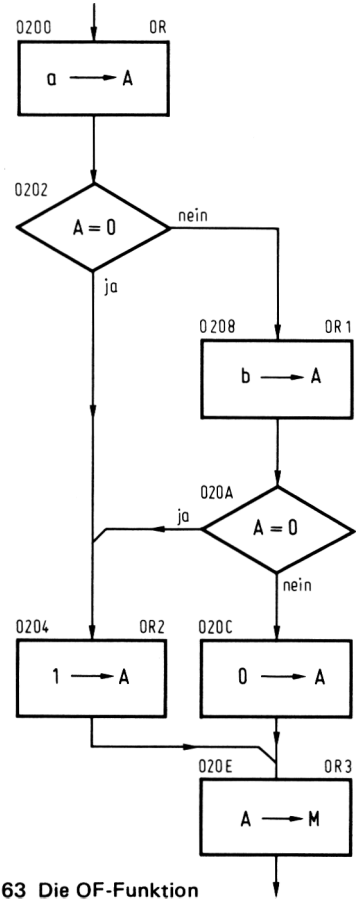


Abb. 63 Die OF-Funktion

Tabelle 4. Programm: EN

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A500	AND	LDA		Z-page a→A
0202	D008		BNE	AND1	Rel. a≠ 0
0204	A501		LDA		Z-page b→A
0206	D004		BNE	AND1	Rel. b≠ 0
0208	A901		LDA		Imm. 1→A
020A	D002		BNE	AND2	Rel.
020C	A900	AND1	LDA		Imm. 0→A
020E	8502	AND2	STA		Z-page A→M
0210					

Um das Programm übersichtlich zu gestalten, folgen die Befehle den senkrechten Linien im Flußdiagramm. Nach dem Befehl für 1 → A auf Adresse 0208 muß ein JMP-Befehl eingegeben werden. Dieser Befehl hat jedoch eine Länge von drei Byte. Da der Akku mit 01 geladen wurde, ist Z aber immer Null, so daß auch ein BNE-Befehl den Sprung hervorrufen kann. Dieser ist nur zwei Byte lang, so daß dadurch ein Speicherplatz eingespart werden kann.

Das Flußdiagramm für die OR-Funktion geht aus *Abb. 63* hervor. Die dazu gehörige Wahrheitstabelle ist:

a = 0	*b = 0*	*M*
0	0	0
0	1	1
1	0	1
1	1	1

In *Tabelle 5* sind die Befehle aufgeführt. Auch hier befindet sich anstelle eines JMP-Befehls ein BNE-Befehl (Adresse 0206). Es ist darauf zu achten, daß im Flußdiagramm unter der Bedingung A = 0 unter b A(OR1) die Angaben „nein" und „ja" im Hinblick auf die anderen Bedingungen in diesem Programm vertauscht sind. Deshalb ist auf Adresse 020A anstelle eines BNE-Befehls ein BEQ-Befehl erforderlich.

Die Befehle für ein Exclusiv-OR-Programm aus *Abb. 64* sind in Tabelle 6 aufgeführt. Hier ist ein JMP-Befehl durch einen BEQ-Befehl auf Adresse 020A ersetzt. Die Wahrheitstabelle für diese Funktion ist wie folgt:

a = 0	*b = 0*	*M*
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 5. Programm: OR

0200	A500	OR	LDA		Z-page a→A
0202	D004		BNE	OR1	Rel. Sprung, wenn a≠0
0204	A901	OR2	LDA		Imm. 1→A
0206	D006		BNE	OR3	Rel.
0208	A501	OR1	LDA		Z-page b→A
020A	F0F8		BEQ	OR2	Rel. Sprung, wenn b=0
020C	A900		LDA		Imm. 0→A
020E	8502	OR3	STA		Z-page A→M
0210					

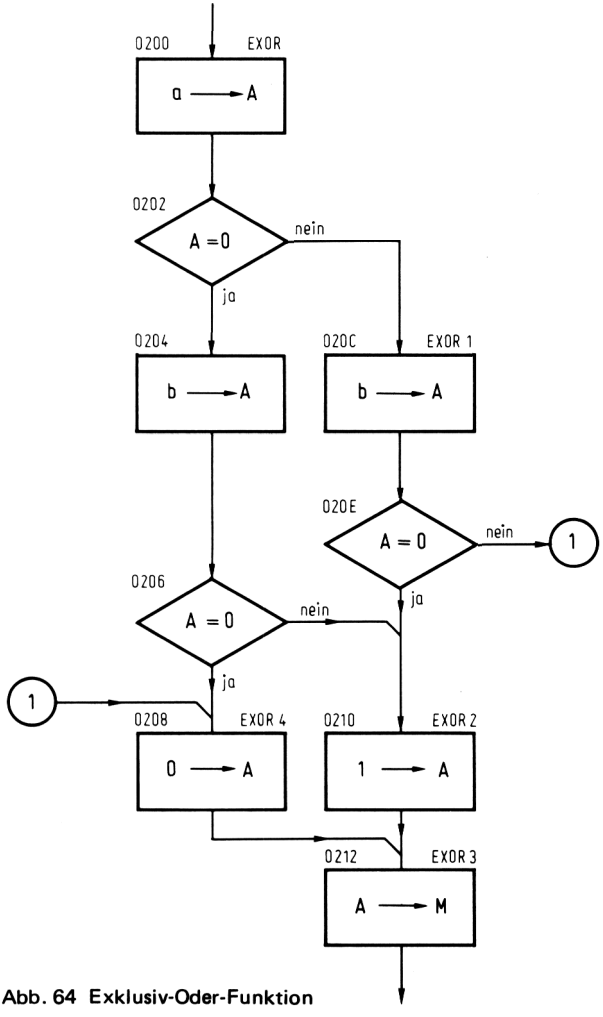


Abb. 64 Exklusiv-Oder-Funktion

Tabelle 6. Programm: EXOR

0200	A500	EXOR	LDA		Z-page a→A
0202	D008		BNE	EXOR1	Rel. a≠0
0204	A501		LDA		Z-page b→A
0206	D008		BNE	EXOR2	Rel. b≠0
0208	A900	EXOR4	LDA		Imm. 0→A
020A	F006		BEQ	EXOR3	Rel.
020C	A501	EXOR1	LDA		Z-page b→A
020E	D0F8		BNE	EXOR4	Rel. b≠0
0210	A901	EXOR2	LDA		Imm. 1→A
0212	8502	EXOR3	STA		Z-page A→M
0214					

13.4 Verzögerungszeiten

Mit einem Mikrocomputer können auch Zeitmessungen ausgeführt werden. Das Prinzip dafür ist einfach. Man läßt während einer zu messenden Zeit ein Programm, dessen Arbeitsgeschwindigkeit bekannt ist, den Computer mehrmals durchlaufen. Die Zahl der Durchläufe während der zu messenden Zeit ergibt die Gesamtzeit. Die Arbeitsgeschwindigkeit eines Programmes kann je nach dem vorliegenden Zeitproblem z.B. 0,01 oder 0,1 s betragen.

Bei einer Programmlaufzeit von z.B. 0,1 s und 15 Durchläufen während der zu messenden Zeit werden 0,1 x 15 = 1,5 s erreicht.

Die Programmlaufzeit ist von der Anzahl der Befehle, die abgearbeitet werden müssen, und den Taktzyklen pro Befehl abhängig. Die Anzahl der Taktperioden pro Befehl ist min. 2 und max. 7. Angenommen, die gemittelte Anzahl pro Befehl beträgt 5 und ein Taktzyklus dauert 10^{-6} s, dann sind für ein Programm von 0,01 s

$$\frac{10^{-2}}{5 \cdot 10^{-6}} = 2000$$

Befehle erforderlich.

Es ist nicht sehr zweckmäßig, lediglich für eine Verzögerung 2000 Befehle in einen Speicher zu laden, ohne daß diese Befehle einen weiteren Zweck erfüllen. Es ist daher besser, das Unterprogramm nach *Abb. 65* hierfür einzusetzen. Hier durchlaufen die Befehle DEX und BNE (x = 0) mehrmals eine Schleife. Für den hier eingesetzten Sprungbefehl BNE sind nach Tabelle 1 zwei Taktzyklen erforderlich. Das gilt aber nur dann, wenn kein Sprung ausgeführt wird. Bei einem Sprung nach einem Speicherplatz auf derselben Seite müssen ein Zyklus, auf eine andere Seite (und zwar nach der nächst-liegenden, weiter weg ist nicht möglich) zwei Zyklen hinzuaddiert werden.

Im vorliegenden Fall wird der Sprung auf derselben Seite ausgeführt. Der BNE-Befehl dauert mit Sprung drei, der DEX-Befehl zwei Zyklen. Zusammen werden für eine Schleife fünf Taktperioden (ein Zyklus ist eine Taktperiode) benötigt.

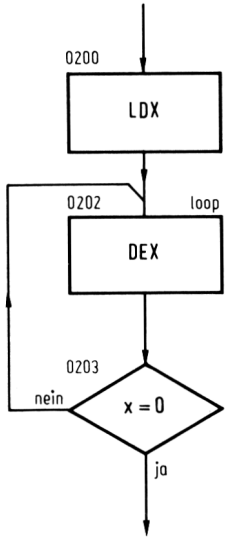


Abb. 65 Verzögerungsschleife mit dem X-Register als Zähler

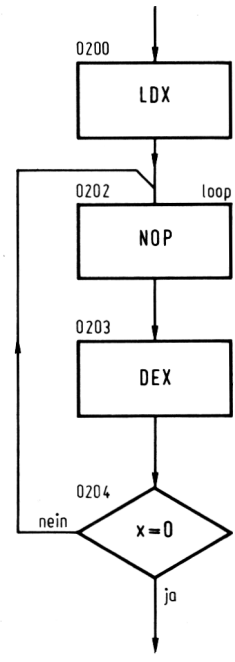


Abb. 66 Verlängerung der Zeit mit dem NOP-Befehl

Wie oft die Schleife durchlaufen wird, ist vom Inhalt des Index-x-Registers (LDX) abhängig. Nennen wir diesen Inhalt x, dann dauert das Programm ohne den LDX-Befehl 5 (x−1) + 4 = 5x−1 Taktperioden. Einmal wird der Sprungbefehl nämlich nicht ausgeführt und benötigt dann nur zwei Zyklen. Dazu kommt noch der LDX-Befehl mit weiteren zwei Zyklen, so daß sich 5x+1 Taktperioden ergeben.

In *Tabelle 7* ist das Programm für x = $ 64 angegeben. Die Gesamtprogrammlänge überstreicht demnach

5 x 100 + 1 = 501

Taktperioden ($ 64 = $100_{(10)}$).
Die max. erreichbare Programmlänge ist:

5 x 256 + 1 = 1281

Taktperioden, da das Index-x-Register max. 00 ≙ 256 speichern kann.

Tabelle 7. Verzögerungszeit

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A264		LDX		Imm. $ 64→x
0202	CA	loop	DEX		Impl. x−1→x
0203	D0FD		BNE	loop	Rel. x≠0
0205					

Tabelle 8. Verzögerungszeit

0200	A264		LDX		Imm. $ 64→x
0202	EA	loop	NOP		Impl.
0203	CA		DEX		x−1→x
0204	D0FC		BNE	loop	Rel. x≠0
0206					

Längere Zeiten können durch Einführen eines besonderen Befehls, wie im Programm nach *Abb. 66* der NOP-Befehl, erreicht werden. Damit können zusätzlich zwei Taktperioden in die Schleife aufgenommen werden, womit sich dann die Programmlaufzeit über

$$7x + 1$$

Taktperioden erstreckt. Auch können andere Befehle, wie INC oder direkte Adressierung, z.B. mit 7 Taktzyklen, eingesetzt werden. Es muß nur darauf geachtet werden, daß Registerinhalte, die nicht geändert werden dürfen, beeinflußt werden. In *Tabelle 8* ist das Programm für das Flußdiagramm nach Abb. 66 aufgelistet.

Die Länge dieses Programms beträgt max. 1786 Taktperioden. Eine besonders große Verzögerungszeit kann mit einer doppelten Schleife (Abb. 67) erreicht werden.

Bei der ersten Schleife ergeben sich 5x + 1 Taktperioden, dazu kommen für DEY und den Sprung nach loop 2 noch 5 Taktperioden, also:

$$5x + 1 + 5 = 5x + 6$$

Für die zweite Schleife ergeben sich

$$n \cdot y + 1$$

Taktzyklen. Hierin bedeutet n die Anzahl der Taktzyklen für einen Durchlauf der zweiten Schleife (n = 5x + 6) und y den Inhalt des Index-y-Registers. Die Anzahl sämtlicher Taktzyklen für dieses Programm ist:

$$(5x + 6)\, y + 1$$

Hiermit wird eine Verzögerung von 326656 Taktzyklen erreicht. Bei einer Taktfrequenz von 1 MHz dauert eine Taktperiode 10^{-6} s. Die max. Programmlaufzeit ist dann 0,326656

Tabelle 9. Verzögerungszeit

0200	A064		LDY		Imm. 64→y
0202	A264	loop 2	LDX		Imm. 64→x
0204	CA	loop 1	DEX		Impl. x−1→x
0205	D0FD		BNE	loop 1	Rel. x≠0
0207	88		DEY		Impl. y−1→y
0208	D0F8		BNE	loop 2	y≠0, Rel.
020A					

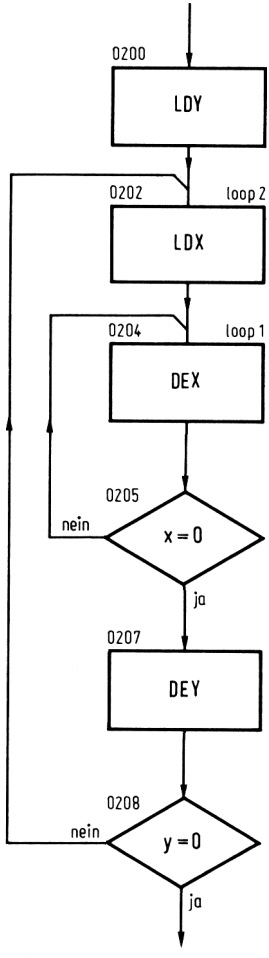


Abb. 67 Verschachtelte Schleife für lange Zeiten mit X- und Y-Registern

s. Eine noch längere Laufzeit läßt sich durch Einführen eines oder mehrerer Befehle in die erste oder zweite Schleife erreichen. Die größte Wirkung wird durch Einführung eines Befehls in die „innere" Schleife erreicht. In *Tabelle 9* sind die Befehle für dieses Programm angegeben.

13.5 Lesen der Eingangstore (Ports)

Die Eingangstore sind Register, die über den Datenbus eingelesen werden können. Sie enthalten stets acht Leitungen, die die Verbindung zwischen Peripheriegerät und Computer bilden. Als Interface zwischen Eingangstore und Peripheriegerät sind Bausteine vorgesehen, die in der Lage sind, die Eingangsleitungen während einer von dem Peripheriegerät bestimmten Zeit auf logisch „0" zu setzen. Die Bausteine können Relaiskontak-

Tabelle 10. Programm: Lies Eingangsleitung 0

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A9FE	Initiiere	LDA		Imm. $ FE
0202	8D0117		STA		Abs. Leitung 0 = Eingang
0205	A901		LDA		Imm. Maske $ 01
0207	2C0017		BIT		Abs. Bestimme Inhalt Leitung 0
020A					

te (u.a. Reedrelais), Schalter, Transistoren, Optokoppler usw. sein. Es können an die Eingangstore aber auch Analog/Digital-Wandler oder SIPO-Schaltungen (seriell in, parallel out) angeschlossen werden.

Alle diese Bausteine haben nur den Zweck, die Eingangsleitungen auf logisch „0" zu setzen, sofern dies erforderlich ist. Sie auf logisch „1" zu bringen, erübrigt sich; denn ohne Steuerung durch ein Bauelement liegt die Eingangsleitung über eine Torschaltung und einen entsprechenden Widerstand nach +u ständig auf logisch „1", dasselbe gilt auch für die nicht im Betrieb befindlichen Eingangsleitungen.

Bei kombinierten Ein-Ausgabetoren kann ein Tor, das durch einen Schreibbefehl als Ausgang geschaltet ist, einen Wert erhalten, der als logisch „0" angesehen werden kann.

Es ist darum immer bei diesen Ein-Ausgangskombinationen erforderlich, eine Leitung, die eingelesen werden soll, zu maskieren.

Bei kombinierten Ein-Ausgabetoren müssen immer die Ausgangsleitungen durch „Initialisierung" definiert werden. Bei den in diesem Buch angegebenen Beispielen wird davon ausgegangen, daß kombinierte Ein-Ausgabetore (peripheral databuffer) die Adresse 1700 und das entsprechende Data-Direction-Register die Adresse 1701 erhalten. Um eine Leitung als Eingangsleitung zu definieren, muß auf der betreffenden Stelle in diesem Datenrichtungsregister eine „0" geschrieben werden. Im Programm der *Tabelle 10* ist die Leitung 0 als Eingangsleitung definiert und wird direkt nach der Initialisierung eingelesen.

Die ersten zwei Befehle laden das Datenrichtungsregister mit $ FE = $11111110_{(2)}$. Dadurch ist die erste Leitung 0 als Eingang definiert. Mit dem dritten Befehl wird die Maske 01 in den Akku eingegeben und der letzte Befehl (BIT) legt den Wert der Leitung 0 fest, ohne daß der Inhalt des Akku verlorengeht. Der Pegel an Leitung 0 ist dann an Z des Statusregisters erkennbar. Ist Z = 0, dann liegt die Leitung auf Pegel „1", bei Z = 1 auf Pegel 0.

In *Abb. 68* kann mit Hilfe von acht Schaltern eine Binärzahl auf die Eingangsleitung gelegt werden. Gewöhnlich wird ein offener Schalter mit „0", ein geschlossener mit „1" definiert. In diesem Fall ist der „H"-Wert eines Eingangstores 0 und der „L"-Wert „1". Um die richtige Zahl zu finden, muß mit einem EXOR-Befehl der invertierte Wert der Eingangstore gelesen werden.

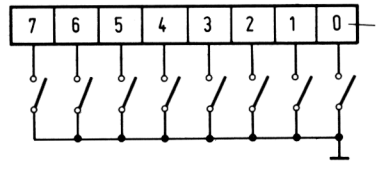


Abb. 68 Abfrage von acht Schaltern über einen Eingangsport

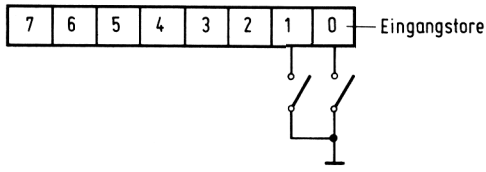


Abb. 69 Zwei Schalter genügen zur Auswahl vier unterschiedlicher Programme

In *Tabelle 11* sind durch die ersten zwei Befehle sämtliche Leitungen als Eingangsleitungen definiert. Dann wird mit Hilfe der Maske $ FF und des EOR-Befehls der Wert der eingegebenen Zahl ermittelt.

Bit	76543210	
Zahl	01011010	
Torwerte	10100101	
Maske	11111111	∀
Akku-Inhalt	01011010	

In *Abb. 69* liegen an den Toren 0 und 1 zwei Schalter, mit denen eine Programmauswahl ausgeführt werden kann. Es ist zwischen zwei Programmen zu wählen, wobei jeder Schalter sein eigenes Programm betätigt. In Abb. 70 ist das Flußdiagramm für das Unterprogramm angegeben, das je nach Schalterstellung auf Programm 1 oder 2 springt.
Wir nehmen an, beide Schalter sind offen. Nach Initialisierung wird der Akku mit $ 01 geladen; das ist die Maske für den darauf folgenden BIT-Befehl. Hierauf wird der Wert von Tor 0 bestimmt:

Tor	76543210	
Torinhalt	xxxxxx11	x = don't care
Maske	00000001	∧
Ergebnis	00000001	Z = 0

Ein Sprung nach Programm 1 findet nicht statt. Der Akku-Inhalt ist nicht geändert, und nach dem ASL-Befehl wird der Inhalt von Tor 1 bestimmt:

Tor	76543210	
Torinhalt	xxxxxx11	x = don't care
Maske	00000010	∧
Ergebnis	00000010	Z = 0

Tabelle 11. Programm: Lies Eingangsleitungen

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A900	Initiiere	LDA		Imm. $ 00
0202	8D0117		STA		Abs. sämtl. Leitungen Eingang
0205	A9FF		LDA		Imm. $ FF Maske
0207	4D0017		EOR		Abs. Bestimme Inversion
020A					

Jetzt wird nun auf „loop" gesprungen und die Schleife nochmals durchlaufen. Dieser Arbeitslauf wird so lange wiederholt, bis ein Schalter geschlossen ist. Wird der Schalter nach Tor 0 geschlossen, dann wird ein Sprung nach Programm 1 ausgelöst. Wird der Schalter nach Tor 1 geschlossen, dann wird am Ende des Unterprogramms nicht nach „loop" gesprungen, sondern es wird mit dem nächsten Befehl als ersten Befehl von Programm 2 fortgefahren.

Dieser Programmablauf ist in *Tabelle 12* wiedergegeben. Nach dem Opcode für den BEQ-Befehl auf Adresse 020A folgt die Zahl n. Diese Zahl gibt die Anzahl von Speicherplätzen an, die vor- oder rückwärts zu springen sind, um Programm 1 zu finden. Pro-

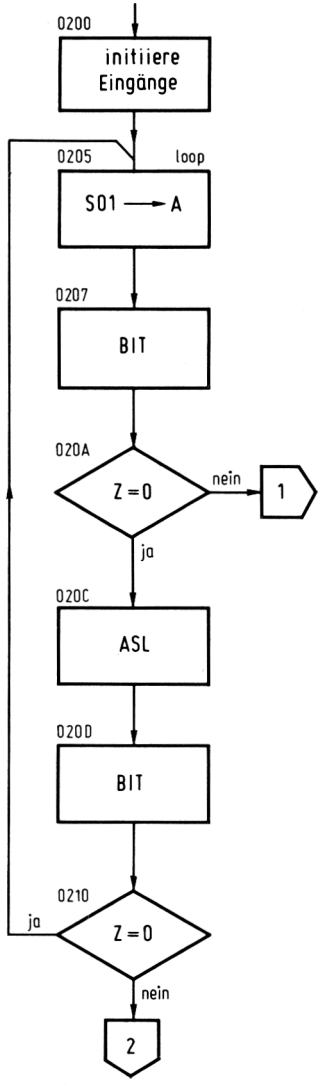


Abb. 70 Mit dem ASL-Befehl in der Schleife werden nacheinander alle Eingangsleitungen maskiert

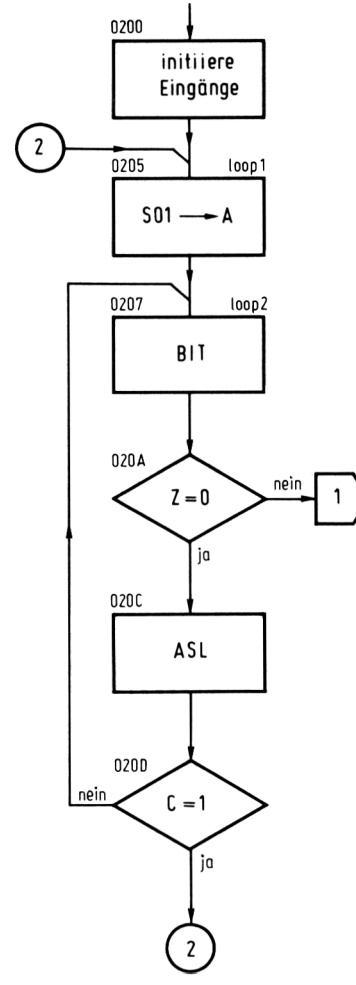


Abb. 71 Hier dient der Inhalt des Carry-Flag als Abbruchbedingung in der Schleife

Tabelle 12. Programm: Lies Leitungen 0 und 1 (wähle Programm)

0200	A9FC	Initiiere	LDA		Imm. $ FC
0202	8D0117		STA		Abs. Tore 0 und 1 Eingang
0205	A901	loop	LDA		Imm. $ 01 Maske
0207	2C0017		BIT		Abs. Tor 0
020A	F0N		BEQ	Progr.1	Rel. Z≠0
020C	0A		ASL		Accu
020D	2C0017		BIT		Abs. Tor 1
0210	D0F3		BNE	loop	Rel.
0212		Programm 2			

Tabelle 13. Programm: Lies Leitungen

0200	A900	Initiiere	LDA		Imm. $ 00
0202	8D0117		STA		Abs. Sämtl. Leitungen Eingang
0205	A901	loop 1	LDA		Imm. $ 01 Maske
0207	2C0017	loop 2	BIT		Abs. Inhalt Tor
020A	F005		BEQ	Prog.	Rel. Z≠0
020C	0A		ASL		Accu
020D	90F8		BCC	loop 2	Rel. C=0
020F	B0F4		BCS	loop 1	Rel. C=1
0211		Prog.			

gramm 2 folgt auf Adresse 0212. Dieses Programm ist auf sämtliche acht Eingangstore und acht verschiedene Programme, auf die gesprungen werden kann, ausdehnbar.

Außerdem ist die Möglichkeit gegeben, nach einem Programm zu springen, wenn eine der acht Eingangsleitungen auf „0" steht (immer das gleiche Programm). Das geht aus dem Flußdiagramm in *Abb. 71* hervor. Da werden nacheinander sämtliche acht Eingangstore auf ihren Inhalt durch „loop 2" untersucht. Nach jedem ASL-Befehl ist das Übertragbit C = 0 und es wird auf loop 2 gesprungen. Sobald das achte Tor untersucht ist, wird C = 1 (das einzige 1-Bit im Akku wird nun durch den ASL-Befehl auf C geschoben). Der nächste Sprung geht auf loop 1. Wieder werden die acht Tore inhaltlich überprüft; das läuft so weiter, bis einer der Schalter geschlossen oder bis ein Tor auf irgendeine andere Art auf „0" gesetzt wird. Danach erfolgt ein Sprung nach dem betreffenden Programm. In *Tabelle 13* ist die Befehlsreihenfolge aufgeführt. Bei diesem Programm ist am Akkuinhalt erkennbar, welche der Leitungen „0" wurde.

13.6 Warteschleifen

Im allgemeinen werden Programme vom Computer in sehr kurzer Zeit abgewickelt. Nach einem Programmdurchlauf gelangt der Computer in eine Warteschleife, um für die Aufnahme neuer Daten bereit zu sein, so daß mit diesen wiederum neue Aufgaben abgearbeitet werden können. Eine oft gebrauchte Warteschleife ist: „Warte auf Eingangstor L",

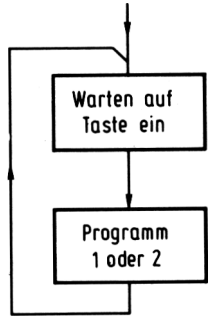


Abb. 72 Warteschleife zum Start eines Programms

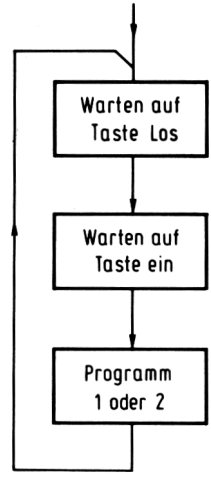


Abb. 73 Wenn das Programm schneller fertig ist, als ein Tastendruck dauert, wird zusätzliche Abfrage nötig

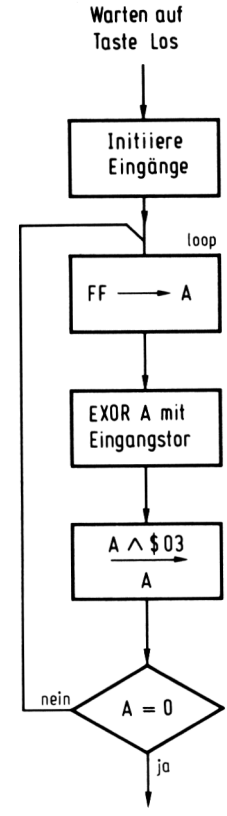


Abb. 74 Flußdiagramm zur Abfrage der Eingangsleitungen

obgleich diese Bezeichnung meistens nicht gebraucht wird. Da die Datenangabe üblicherweise über ein Tastenfeld erfolgt, wird vielfach die Bezeichnung „Warte auf Taste ein" gebraucht.

Wir haben bereits bestimmte Arten von Warteschleifen kennengelernt. In den in Abb. 70 und 71 dargestellten Programmen wird jeweils eine Schleife durchlaufen, bis einer der Schalter (Tasten) geschlossen ist. Dann wird auf ein bestimmtes Programm gesprungen. Auch das Programm in Abb. 16 enthält einen Wartezyklus. Der Unterschied zu dem vorhergehenden liegt jedoch darin, daß die auszuführenden Vorgänge aufgrund einer bestimmten Schalterstellung innerhalb des Zyklus vorgenommen werden und nicht auf ein Programm übergegangen wird (Bedienung der Signallampen).

Im Programm nach Abb. 70 muß beim Drücken der Taste 1 nach Programm 1 gesprungen werden. Im allgemeinen muß nach dem Durchlauf von Programm 1 gewartet werden, bis wieder eine Taste gedrückt wird. Dann erfolgt wieder ein Sprung auf das gewählte Programm, darauf wieder warten usw., wie im Flußdiagramm (*Abb. 72*) angegeben.

Normalerweise dauert das Drücken einer Taste wesentlich länger als ein Programmdurchlauf. Kommt das Programm beim „Warten auf Taste ein" zurück, dann ist die Taste noch gedrückt und das Programm wird nochmals durchlaufen. Es kann vorkommen, daß ein Programm nicht mehr als einmal durchlaufen werden darf. Dann muß ein weiterer Wartezyklus, nämlich „Warten bis Taste los" eingefügt werden (siehe Flußdiagramm Abb.

74). Nach Durchlauf des gewählten Programms wird auf Zyklus „Warten bis Taste los" gesprungen und sobald die Taste losgelassen wird, kommen wir auf Zyklus „Warten auf Taste ein" (*Abb. 73*).

Wir gehen vom Wartezyklus in Abb. 70 aus. Angenommen, die Eingangsleitung 0 liegt durch das Eindrücken einer Taste auf Pegel L, so daß das Unterprogramm 1 durchlaufen wird, dann kommen wir in den Wartezyklus (*Abb. 74*), während die Tore auf xxxxxx10 liegen. Nach EXOR-Befehl finden wir im Akku: xxxxxx01. Jetzt folgt EN-Auftrag mit Maske $ 03:

```
A       xxxxxx01
Maske   00000011 ∧
        --------
A       00000001  Z = 0
```

Mit einem BNE-Befehl wird nun zurück auf „loop" gesprungen.

Liegt der Leitungseingang 1 durch Drücken der anderen Taste auf Pegel L, dann wird Programm 2 durchlaufen und es ergibt sich an den Eingangstoren der Wert

xxxxxx01.

Nach dem EXOR-Befehl ist nun der Akku-Inhalt:

xxxxxx10

Der AND-Auftrag mit Maske $ 03 ergibt:

```
A       xxxxxx10
Maske   00000011 ∧
        --------
A       00000010  Z = 0
```


Nun wird auf „loop" gesprungen.

Wird die gedrückte Taste losgelassen, liegen beide Eingangsleitungen auf Pegel H und die Eingangstore haben den Wert:

xxxxxx11.

Nach EXOR-Befehl ergibt sich der Akku-Inhalt zu:

xxxxxx00

und der NE-Auftrag mit Maske $ 03:

```
A       xxxxxx00
Maske   00000011 ∧
        --------
A       00000000  Z = 1
```

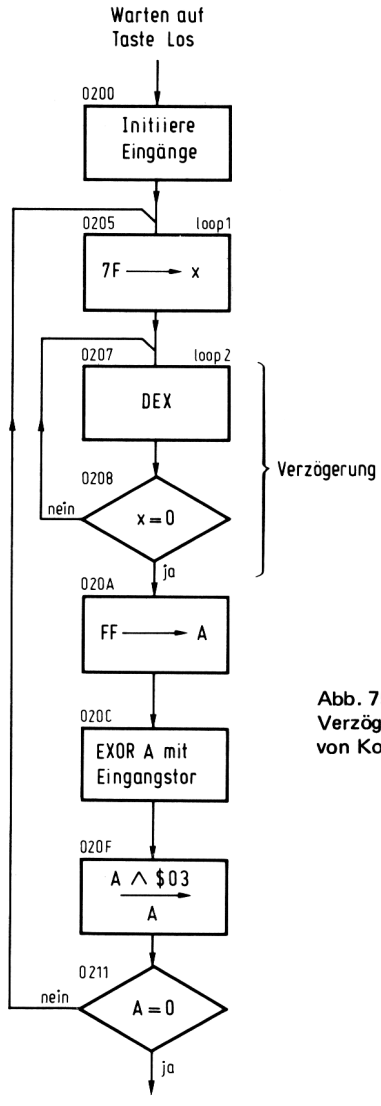


Abb. 75 Verbessertes Flußdiagramm mit Verzögerungsschleife zum Vermeiden von Kontaktprellen

Da jetzt kein Sprung stattfindet, wird das Unterprogramm verlassen. Das Unterprogramm nach Abb. 70 wird anschließend abgearbeitet, bis eine der Tasten gedrückt wird (Warten auf Taste ein).

Ein ungewünschter Nebeneffekt bei Drucktasten und Schaltkontakten ist das sog. Kontaktprellen. Dieses hat zur Folge, daß während der kurzen Zeit des Drückens eine ganze Reihe von Impulsen ausgelöst wird, die das gewählte Programm mehr als einmal durchlaufen lassen. Diese Nebenwirkung kann durch eine an den Zyklus „Warten auf Taste los" angekoppelte Verzögerungsschleife verhindert werden. Ist die Verzögerung groß genug, kann festgestellt werden, ob die Taste noch gedrückt ist, wenn das Prellen

Tabelle 14. Programm: Warten auf Taste los

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A9FC	Initiiere	LDA		Imm. $ FC
0202	8D0117		STA		Abs. Tore 0 und 1 Eingang
0205	A27F	loop 1	LDX		Imm. $ 7F
0207	CA	loop 2	DEX		Impl. } Verzögerung
0208	D0FD		BNE	loop 2	Rel. } Verzögerung
020A	A9FF		LDA		Imm. $ FF Maske
020C	4D0017		EOR		Abs. Eingangstor
020F	2903		AND		Imm. $ 03 Maske
0211	D0F2		BNE	loop 1	Rel.
0213					

abgeklungen ist. das vollständige Flußdiagramm geht aus *Abb. 75* hervor und die Befehlsfolge aus *Tabelle 14.*

Es wird darauf hingewiesen, daß am Zyklusanfang „Warten auf Taste ein" eine Initialisierung überflüssig ist.

13.7 Parallel-Serienumsetzung

Der Datentransport nach dem Peripheriegerät erfolgt zumeist seriell, das besagt, daß die Bits eines Binärwortes zeitlich nacheinander auf die Ausgangsleitungen gegeben werden. Meistens sind diese Bits dann Elemente von Codeworten (ASCII) mit folgenden Eigenschaften:

Einschließlich eventueller Paritätsbits haben sämtliche Worte eines bestimmten Systems die gleiche Anzahl von Elementen.
Sämtliche Wortelemente sind gleich lang.

Das Peripheriegerät muß über Beginn eines Codewortes und Länge der Codewort-Elemente informiert werden. Hierzu stehen zwei Übertragungsmöglichkeiten zur Verfügung: Synchrone bzw. asynchrone Übertragung.
Für die synchron-serielle Steuerung von Codeworten sind drei Signale erforderlich:

a) Ein Taktsignal für den Codeelement-Anfang.
b) Ein Signal für das eigentliche Codewort.
c) Ein Signal für den Codewort-Anfang.

In *Abb. 76* sind diese Signale wiedergegeben. Es wird davon ausgegangen, daß die abfallenden Flanken der Signale a und c zur Markierung dienen. Das ist allgemein gebräuchlich, da die Leitungen im Ruhezustand auf H-Pegel liegen.

Das Signal b stellt das Codewort 01101001 dar. Es ist darauf zu achten, daß B0 der Signalfolge b nur daran erkennbar ist, daß das Signal c den Codewort-Anfang kennzeichnet.

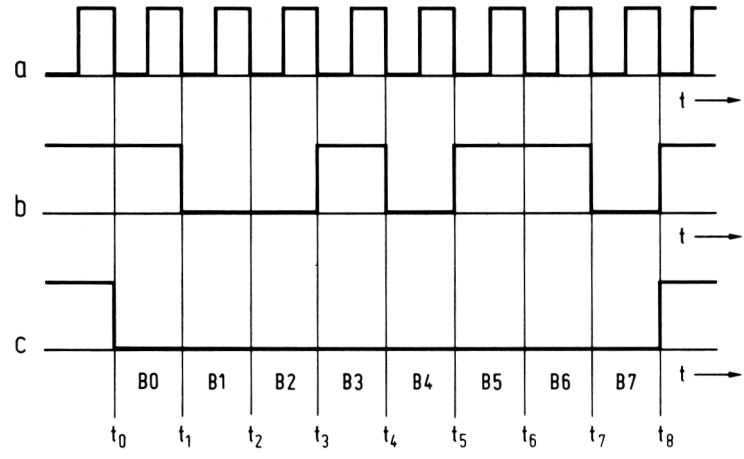


Abb. 76 Serielle synchrone Übertragung eines Byte

Das Umsetzen der parallelen Daten der Ausgangstore zu einem seriellen Bitstrom auf den Sammelleitungen zum Peripheriegerät wird im allgemeinen bei einem Synchronsystem durch entsprechende Hardware ausgeführt.

Beim asynchronen Steuern von Codeworten läuft nur das Signal zum Peripheriegerät, was dem Codewort entspricht, und es gibt keine besondere Taktleitung. Den Anfang des Codewortes kennzeichnet ein sog. Startbit. Da im Ruhezustand die Leitung stets auf H-Pegel liegt, ist das Startbit grundsätzlich ein L-Bit. Nach dem Startbit folgt das Codewort, abgeschlossen durch ein Stophit. Dieses kann auch länger sein als ein Datenbit und liegt stets auf H-Pegel.

Im Peripheriegerät werden die eingehenden Bit auf ihren Wert abgetastet, wobei der Abtastpunkt mit der betreffenden Bitmitte zusammenfällt.

In *Abb. 77* ist ein Fünf-Bit-Codewort mit vorangehendem Startbit (Bit 0) und abschließendem Stopbit (Bit 6 und Bit 7) dargestellt. Auch sind die Abtastpunkte angegeben. Hierfür muß das Peripheriegerät selbst ein Taktsignal liefern, dessen zeitliche Periode so zu wählen ist, daß sämtliche Abtastpunkte in das Zeitintervall des betreffenden Codebits fallen.

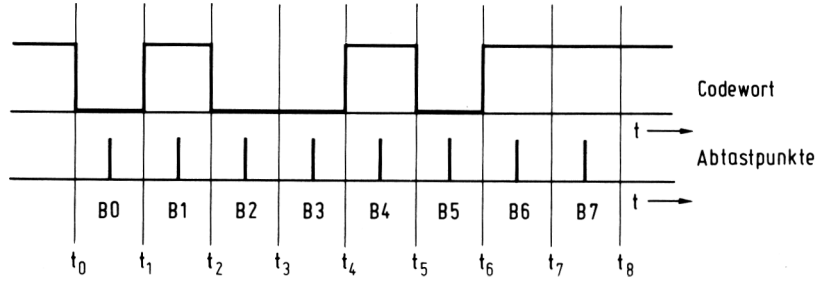


Abb. 77 Asynchrone Übertragung eines 5-Bit-Wortes (z.B. bei einem Baudot-Fernschreiber)

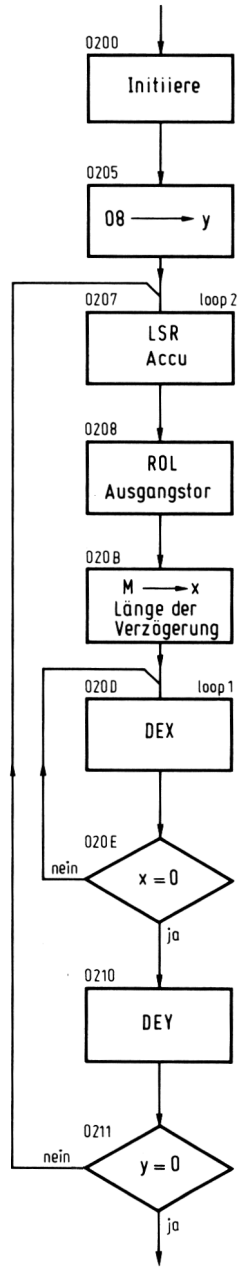


Abb. 78 Parallel/Serienumsetzung per Software

Auch das asynchrone serielle Steuern von Codeworten kann durch Hardware erfolgen. Ein Programm für software-Steuerung ist im Flußdiagramm (*Abb. 78*) aufgezeichnet. Hier wird davon ausgegangen, daß am Programmanfang das betreffende Codewort mit Start- und Stopbit in den Akku geladen wird. Vom Ein/Ausgabeport wird P0 Aus-

Tabelle 15. Programm: Seriell aus

0200	A201	Initiiere	LDX		Imm. $ 01
0202	8E0117		STX		Abs. Tor 0 Ausgang
0205	A008		LDY		Imm. $ 08 für acht Bit
0207	4A	loop 2	LSR		Accu
0208	2E0017		ROL		Abs.
020B	A600		LDX		Z-page
020D	CA	loop 1	DEX		Impl. } Verzögerung
020E	D0FD		BNE	loop 1	Rel. } Verzögerung
0210	88		DEY		Impl.
0211	D0F4		BNE	loop 2	Rel.

gang. Mit LSR-Befehl wird B0 aus dem Akku nach C geschoben. Der nächste ROL-Befehl schiebt C nach P0 vom Ausgangstor. Die anschließende Verzögerung läßt das ausgeschobene Bit genügend lange auf der Ausgangsleitung stehen. In der nächsten Tabelle ist die Zahl der Taktzyklen (M = $197_{(10)}$) angegeben:

	Takt-Zyklen
LSR Akku	2
ROL direkt	6
LDX zero page	3
DEY implizit	2
BNE relativ	3
Verzögerung	
197 x 5 – 1 =	984
	1000

Eine ganze Schleife wird in 1000 Taktzyklen durchlaufen, das bedeutet bei einer Taktfrequenz von 10^6 Hz eine Arbeitsgeschwindigkeit von 1 ms. Eine Zeitverzögerung – im vorliegenden Fall 197 x 5 – 1 = 984 – ist durch Eingeben einer Zahl ($197_{(10)}$ = S C 5) in Speicherplatz 0000 einstellbar. Das gesamte Programm wird acht Mal durchlaufen, bis nacheinander alle acht Bits aus dem Akku in das Ausgangstor P0 geschrieben sind. In *Tabelle 15* ist die Befehlsfolge dargestellt.

13.8 Pufferspeicher

Es kann vorkommen, daß in den Computer Daten eingegeben werden sollen, jedoch noch nicht verarbeitet werden können und daher eine gewisse Zeit gespeichert werden müssen. Kommen dabei ständige neue Daten hinzu, die in der Reihenfolge ihrer Eingabe verarbeitet werden müssen, erweist sich ein Pufferspeicher als erforderlich.

Das Laden des Speichers läuft, wie in *Abb. 79* gezeichnet, ab. Die erste eintreffende Information wird auf Adresse 0000 eingegeben (A1). Ein „Zeiger" gibt als nächsten lee-

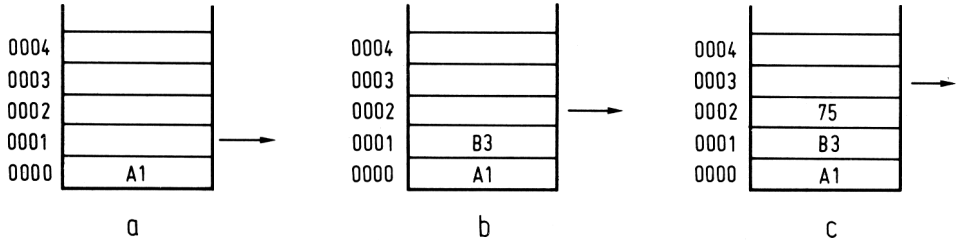


Abb. 79 Wirkungsweise eines Pufferspeichers

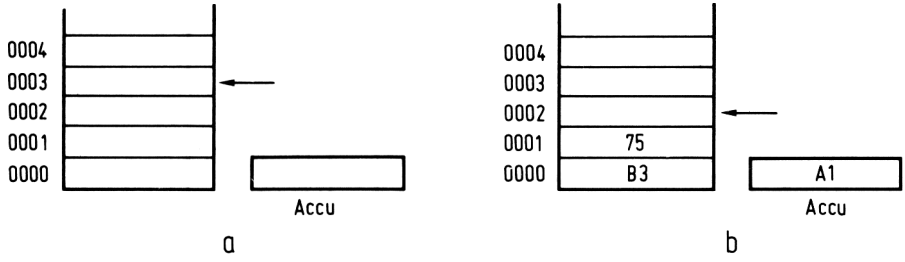


Abb. 80 Speicher- und Akkuinhalt vor und nach dem Einlesen eines Byte in den Puffer

ren Speicherplatz 0001 an. In diese wird die zweite Information (B3) eingegeben und der Zeiger geht nach 0002, und so fort mit den weiteren Daten.

Beim Abarbeiten der Daten wird immer der Inhalt von Speicherplatz 0000 in den Akku geladen und dann alle Daten auf einen tieferen Speicherplatz zurückgesetzt. Aus den *Abbildungen 80a* und *80b* ist der Zustand vor bzw. nach der Operation dieser beiden Vorgänge ersichtlich.

Da der Speicherumfang eines Mikrocomputers nicht unbegrenzt ist, wird auf diese Weise der Speicherplatzbedarf für den Puffer so klein wie möglich gehalten. Desweiteren muß die mittlere Datenverarbeitungsgeschwindigkeit mindestens gleich der der Dateneingabe sein.

In *Abb. 81* ist das Flußdiagramm für das Programm zum Laden des Puffers angegeben. Für den Zeiger ist Speicherplatz 0100 bereitgestellt. Indem dieser in das Index-x-Register geladen wird, kann mit Zero-Page-x-Adressierung der Akku-Inhalt (die Daten) auf eine leere Stelle des Pufferspeichers geschrieben werden. Mit einem INC-Befehl wird der Zeiger um 1 inkrementiert. Die Befehle für dieses Programm sind in *Tabelle 16* aufgelistet. Das Flußdiagramm des Programms zum Lesen des Puffers geht aus *Abb. 82* hervor.

Danach wird zuerst der Speicherplatz 0000 gelesen. Da der Akku für das Nachschieben des Speicherplatzinhaltes vom Puffer benötigt wird, ist der Akku-Inhalt mit PHA-Befehl in den Kellerspeicher zu übertragen. Nun wird mit Zero-Page-x-Adressierung der Speicherplatz 0001 ausgelesen (x = 0) und auf 0000 überschrieben. Nach Inkrementierung von x

Tabelle 16. Programm: Laden des Puffers

0200	AE0001	LDX	point.	Abs. pointer→x
0203	9500	STA		Z-page, x data in buffer
0205	EE0001	INC	point.	Point. + 1→point.
0208				

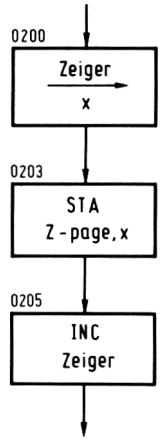


Abb. 81 Flußdiagramm zum Laden des Puffers

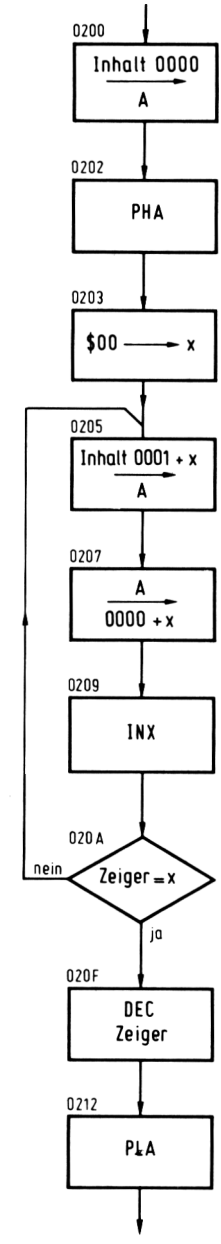


Abb. 82 Auslesen des Pufferspeichers

(x = 1) und Vergleich mit dem Zeigerinhalt wird nach „loop" gesprungen, sofern der Zeiger einen größeren Inhalt als das Index-x-Register hat. Nunmehr wird die Adresse 0002 ausgelesen und in 0001 eingegeben. Das wird so oft fortgesetzt, bis der Inhalt des Index-x-Registers mit dem des Zeigerregisters übereinstimmt. Der letzte Speicherplatz ist dann bearbeitet, mit der Dekrementierung des Zeigers um 1 und dem Holen des Akku-Inhaltes

Tabelle 17. Einlesen eines Puffers

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A500	Read	LDA		Z-page Daten→A
0202	48		PHA		Impl. A→M_s, rette A
0203	A200		LDX		Imm. $ 00
0205	B501	loop	LDA		Z-page, x (00, 01+x)→A
0207	9500		STA		Z-page, x A→(00, 00+x)
0209	E8		INX		Impl.
020A	EC0001		CPX	point	Abs. x-Pointer
020D	D0F6		BNE	loop	Rel.
020F	CE0001		DEC	point	Abs. Pointer−1
0212	68		PLA		
0213					

ist das Programm abgewickelt. Damit ist einmalig eine Information aus dem Puffer gelesen. Das Programm ergibt sich aus *Tabelle 17.*

13.9 Das Siebensegment-Display

Es ist bereits mehrmals von einem Datentransport in Codeworten, z.B. ASCII, die Rede gewesen, wobei dann eine Tabelle zum Umsetzen von Hexadezimalworten in den gewünschten Code erforderlich war. In einem nachfolgenden Beispiel wird das Aufleuchten eines Siebensegment-Display an Hand einer Tabelle demonstriert.

Man benötigt hierzu zwei I/O-Ports; einen zum Eingeben der Hexadezimalzahlen und einen weiteren für die Verbindung zum Display. Das letzte Register wird auf Adresse 1700 als Ausgang geschaltet und mit „Tore A" (PA0 bis einschließlich PA7) bezeichnet. Zum Einlesen der Hexadezimalzahlen gehen wir von einem Register auf Adresse 1702 aus, das, mit der Kennzeichnung „Tore B" (PB0 bis einschließlich PB7), als Eingang geschaltet ist. Vier dieser Eingangstore (PB0 bis einschl. PB 3) werden mit Schaltern (Abb. 83) verbunden, mit denen die Hexadezimalzahlen binär eingegeben werden können, und zwar mit den Werten $0000_{(2)}$ bis einschl. $1111_{(2)}$. Das Data-Direction-Register für die Tore B hat die Adresse 1703.

Auf sieben Ausgangstoren (PA0 bis einschl. PA6) erscheint eine binäre Zahl, wodurch das hier angeschlossene Siebensegment-Display die eingegebene Zahl in hexadezimalen Zeichen wiedergibt.

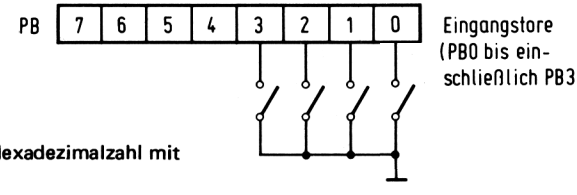


Abb. 83 Eingabe einer 4-Bit-Hexadezimalzahl mit vier Schaltern

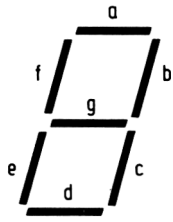


Abb. 84 Bezeichnung der Segmente bei einer Siebensegment-Anzeige

In *Abb. 84* sind die sieben Segmente mit den Buchstaben a bis g gekennzeichnet. In dieser Reihenfolge sind sie mit den Ausgangstoren PA0 bis einschl. PA6 verbunden. Im allgemeinen ist ein direkter Anschluß der Segmente an die Tore nicht möglich. Die Zwischenschaltung eines Puffers ist vielmehr erforderlich. Unter Zugrundelegung, daß das betreffende Segment bei H-Pegel des Ausgangstores aufleuchtet, kann die *Tabelle 18* aufgestellt werden.

Das besagt, daß, wenn Tor 0 auf H-Pegel liegt, Segment a, Tor 1 auf H liegt, Segment b aufleuchtet. Der Tabelle ist außerdem zu entnehmen, welche binäre Zahl am Eingangstor liegen muß, damit das Display eine bestimmte Zahl wiedergeben kann.

Zum Codieren eignet sich besonders die indizierte Adressierung. Dabei unterstellen wir, daß sich die Werte in der Tabelle für den Siebensegment-Code auf den Speicherplätzen 0300 bis einschl. 030F, und zwar in der Reihenfolge 0300 $ 3F, 0301 $ 06 usw. bis 030F $ 71 befinden. Weiter nehmen wir an, daß im Speicherplatz 020D der Operationscode des (indirekten) LDA-Befehls y und zugleich auf 020E $ 00 geladen ist. In den Spei-

Tabelle 18.

Eingang		*Siebensegment*	
binär	hex	binär 76543210 gfedcba	hex Tor Segment
0000	0	00111111	3F
0001	1	00000110	06
0010	2	01011011	5B
0011	3	01001111	4F
0100	4	01100110	66
0101	5	01101101	6D
0110	6	01111101	7D
0111	7	00000111	07
1000	8	01111111	7F
1001	9	01101111	6F
1010	A	01110111	77
1011	B	01111100	7C
1100	C	00111001	39
1101	D	01011110	5E
1110	E	01111001	79
1111	F	01110001	71

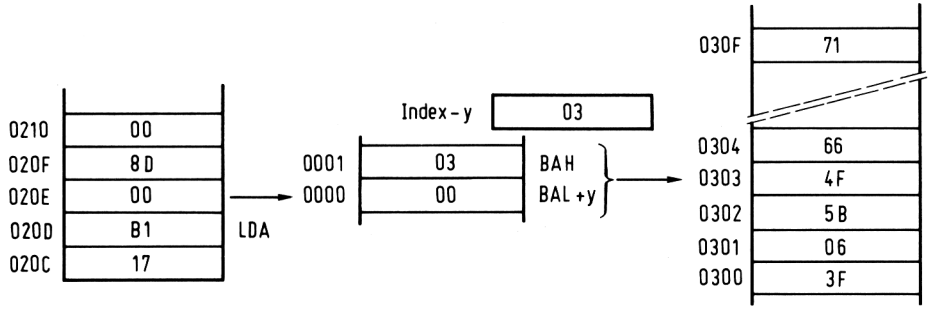


Abb. 85 Codewandlung mit indizierter Adressierung

cherplätzen 0000 und 0001 steht dann die Basisadresse 0300 (*Abb. 85*). Wenn die Eingangstore die Zahl 03 anzeigen, wird diese in das Index-y-Register eingegeben. Nun wird die Adresse 0303 (BAH, BAL + y; 0300 + 03 = 0303) gelesen. Auf dieser Adresse steht der Siebensegment-Code für die Ziffer 3. Dieser wird nun in den Akku gespeichert.

Für dieses Programm sind Flußdiagramm in *Abb. 86* und Befehlsfolge in *Tabelle 19* angegeben.

Beim Initialisieren werden die Tore PB0 bis einschl. PB 3 als Ein-, PB4 bis einschl. PB7 als Ausgang geschaltet (Adresse 0207). Das entsprechende Daten-direction-register erhält die Adresse 1703. Hier wird davon ausgegangen, daß beim Lesen der Adresse 1702 die Ausgangstore auf logisch „0" liegen.

Nun wird direkt die Zahl 03 ausgelesen, da die zwei ersten Schalter auf logisch „1", der andere auf logisch „0" liegen (hier ist ein offener Schalter „1", ein geschlossener „0").

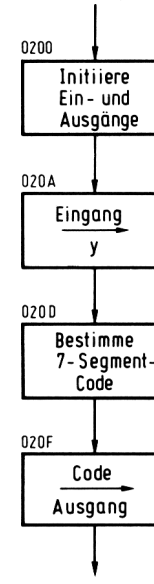


Abb. 86 Ausgabe einer 4-Bit-Zahl auf das Siebensegment-Display

Tabelle 19: Siebensegment-Codierung

0200	A97F	Init.	LDA	Imm. $ 7F
0202	8D0117		STA	Abs. PA0 bis PA6 Ausgang
0205	A9F0		LDA	Imm. $ F0
0207	8D0317		STA	Abs. PB0 bis PB3 Eingang
020A	AC0217		LDY	Abs. Eingang→y
020D	B100		LDA	(indirect), y Siebensegment-Code
020F	8D0017		STA	Abs. 7-segm. code→Ausgang
0212				

Mit einem LDY-Befehl wird der durch den Schalter bestimmte Wert in das Index-y-Register eingegeben. Der weitere Programmverlauf ist leicht zu verfolgen. Soll ein offener Schalter mit „0" und ein geschlossener mit „1" beaufschlagt werden, dann müssen sämtliche acht Tore PB als Eingang geschaltet sein. Für die Zahl 3 ergeben sich dann die Tore:

11111100

(PB0 und PB1 sind „0", da die betreffenden Schalter geschlossen sind.) Mittels EOR-Befehl mit Maske $FF ist eine Inversion anzusetzen.

Tore	11111100
Maske	11111111 ∀
Ergebnis	00000011

Der Programmverlauf gleicht dem der *Tabelle 19.*

13.10 Multiplikation

Addition und Subtraktion sind die von einer CPU direkt durchführbaren arithmetischen Aufgaben. Im Prinzip kennt die CPU nur die Addition. Obgleich ein Subtraktionsbefehl zur Verfügung steht (SBC), wird die Subtraktion intern auf eine Addition durch Bilden des Zweier-Komplementes vom Subtrahenden zurückgeführt.

Das bedeutet jedoch nicht, daß ein Computer keine anderen arithmetischen Aufgaben lösen kann. Für diese müssen dann besondere Programme, gegebenenfalls als Unterprogramm, aufgestellt werden.

Auch für eine Multiplikation ist ein Programm zu erstellen. Betrachten wir hierzu die Multiplikation der Zahlen 1101 und 1011:

1101	Multiplikand
1011	Multiplikator
1101	Zwischenergebnisse
1101.	
1101.	
10001111	Ergebnis.

Daraus ist folgendes zu schließen:

Das Zwischenergebnis wird durch Verschieben des Multiplikanden um einige Stellen nach links erhalten. Um wieviele Stellen verschoben werden muß, hängt vom Stellenwert des entsprechenden Multiplikatorbit ab. Bei einem Vier-Bit-Multiplikator wird um max. drei Bit geschoben. Bei einem Acht-Bit-Multiplikator um max. sieben Bit, usw.

Das „Ergebnis" kann um ein Bit größer sein als das größte Zwischenergebnis. Das Ergebnis von zwei 8-Bit-Zahlen erfordert daher eine 16-Bit-Registerlänge. Das maximale

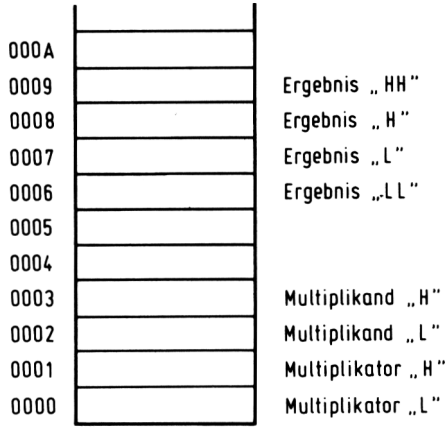


Abb. 87 Speicherverteilung bei der Multiplikation

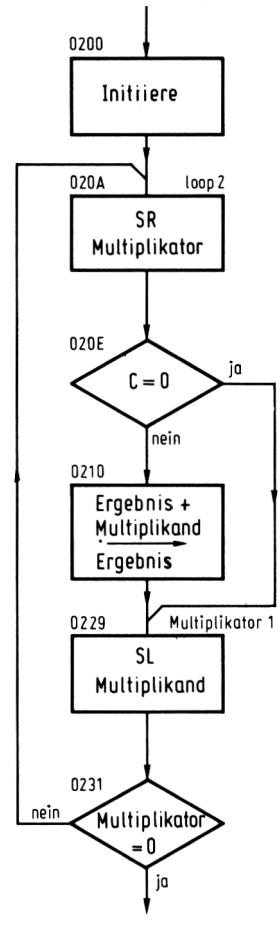


Abb. 88 Multiplikation von zwei 16-Bit-Zahlen

Zwischenergebnis ergibt sich dann zu: 7 + 8 = 15 Bit (7 Bit zum Schieben). Das „Ergebnis" ist ein Bit größer. Das Produkt zweier 16-Bit-Zahlen erfordert eine Registerlänge von 32 Bit.

Im nächsten Programm für eine Multiplikation wird von zwei 16-Bit-Zahlen ausgegangen. Die 2-Bit-Blöcke benötigen vier Speicherplätze; hierfür stehen die Speicherplätze 0000 bis 0003 zur Verfügung. Davon zwei für den Multiplikator und zwei für den Multiplikanden. Das Multiplizieren ist austauschbar ($a \cdot b = b \cdot a$); darum ist ohne Bedeutung, welche Zahl mit Multiplikator und welche mit Multiplikand bezeichnet wird.

Wir nehmen an, daß in den Speichern 0002 und 0003 der Multiplikand gespeichert ist. Da der Multiplikand, um das Zwischenergebnis zu finden, verschoben werden muß, sind nochmals zwei Speicherplätze, 0004 und 0005 erforderlich. Für das Ergebnis sind die Speicherplätze 0006 bis 0009 vorgesehen (*Abb. 87*).

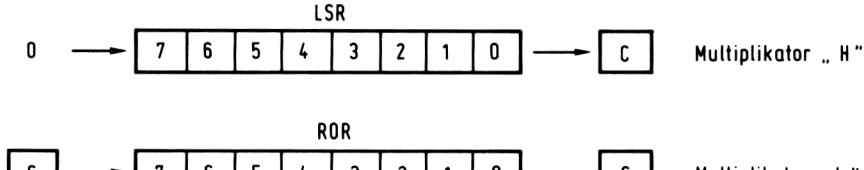


Abb. 89 Schieben des Multiplikators nach rechts

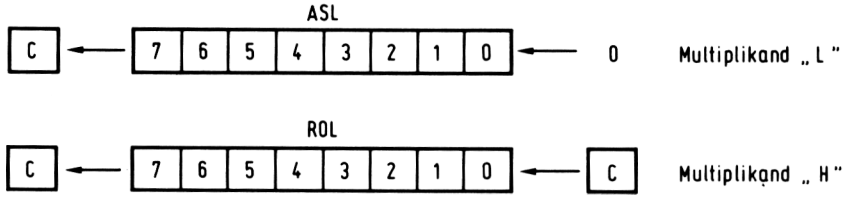


Abb. 90 Schieben des Multiplikanden nach links

Das Flußdiagramm für das Programm geht aus *Abb. 88* hervor. Zuerst müssen die Speicherplätze 0004 bis 0009 mit $ 00 beschrieben werden (initiieren). Nach der Initialisierung wird der Multiplikator eine Stelle nach rechts geschoben. Dazu muß der Inhalt der zwei Register ebenfalls geschoben werden.

Nach dem Schiebebefehl LSR wird zuerst der Multiplikator auf H' und dann mit Schiebebefehl ROR der Multiplikator auf L' gelegt. Dann gelangt der Inhalt von B0 aus Multiplikator H' über C nach B7 von Multiplikator L'. Den niedrigsten Bit-Wert aus Multiplikator L' finden wir endlich in C (*Abb. 89*).

War der niedrigste Bitwert des Multiplikators 1 und ist daher C = 1, dann ist das erste Zwischenergebnis gleich dem Multiplikand. Man beachte hierzu das angegebene Multiplikationsbeispiel mit den Zahlen 1101 und 1011. Der Multiplikand muß nun in das Ergeb-

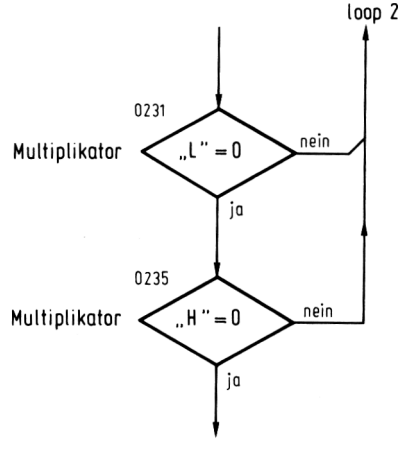


Abb. 91 Der Null-Test dient als Abbruchbedingung der Multiplikation

Tabelle 20: Multiplikation

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	D8	Init	CLD		Impl.
0201	A900		LDA		Imm. $ 00
0203	A206		LDX		Imm. $ 06
0205	9503	loop 1	STA		Z-page, x } Reg. 0004
0207	CA		DEX		Imp. } bis 0009
0208	D0FB		BNE	loop 1	Rel. } werden $ 00
020A	4601	loop 2	LSR		Z-page Multiplikator H
020C	6600		ROR		Z-page Multiplikator L
020E	9019		BCC	Multiplikator 1	Rel. C=0 keine Addition
0210	18		CLC		Impl.
0211	A502		LDA		Z-page
0213	6506		ADC		Z-page
0215	8506		STA		Z-page
0217	A503		LDA		Z-page
0219	6507		ADC		Z-page Ergebnis
021B	8507		STA		Z-page +
021D	A504		LDA		Z-page Multiplikand
021F	6508		ADC		Z-page →
0221	8508		STA		Z-page Ergebnis
0223	A505		LDA		Z-page
0225	6509		ADC		Z-page
0227	8509		STA		Z-page
0229	0602	Multiplikator 1	ASL		Z-page
022B	2603		ROL		Z-page SL
022D	2604		ROL		Z-page Multiplikand
022F	2605		ROL		Z-page
0231	A500		LDA		Z-page Multiplikator L } Multi-
0233	D0D5		BNE	loop 2	Rel. } plikator
0235	A501		LDA		Z-page Multiplikator H } = 0
0237	D0D1		BNE	loop 2	Rel.
0239					

nisregister eingegeben werden, und zwar nach dem Prinzip 0 + A = A. Der Multiplikand wird zum Ergebnis, das außerdem noch den Wert Null hat, hinzuaddiert.

Übereinstimmend mit dem, was in Abschnitt 10.3 über die Addition von zwei Worten, die größer als ein Byte sind, gesagt wurde, muß hier die Addition viermal wiederholt werden. Es müssen daher viermal zwei 8-Bit-Zahlen zusammenaddiert werden, plus ein Carry-Bit mit seinem Wert aus der vorhergehenden Addition.

War das letzte Multiplikator-Bit Null und ist daher C = 0, dann ist der Wert des ersten Zwischenergebnisses noch nicht gefunden und eine Addition zum „Ergebnis" findet nun nicht statt.

Mit dem jetzt folgenden Verschieben des Multiplikanden nach links (SL Multiplikand) kann ggf. das nächste Zwischenergebnis gebildet werden. Der Multiplikand ist nun mit dem Faktor $2_{(10)}$ multipliziert. Zuvor wird auf das erste Byte (Multiplikator L') ein ASL-Befehl und dann auf die höheren Byte des Multiplikanden ein ROL-Befehl angewandt (*Abb. 90*). Hierbei wird auch auf die Register 0004 und 0005 Bezug genommen, so daß durch den Schiebevorgang kein Multiplikanden-Bit verlorengehen kann. Nun wird der Vorgang wiederholt.

Ist das nächste Multiplikator-Bit 1, dann enthält der Multiplikand ein Zwischenergebnis, das zum „Ergebnis" addiert werden kann. Ist das Multiplikator-Bit 0, dann muß der Multiplikand nochmals nach links geschoben werden usw. Das Programm wiederholt sich, bis sämtliche Multiplikator-Bit abgearbeitet sind. Der Multiplikator ist dann Null und die Multiplikation ist abgeschlossen. Aus *Abb. 91* geht hervor, daß sowohl Multiplikator „H" als auch Multiplikator „L" inhaltlich untersucht werden müssen. Sind beide Null, dann ist die Multiplikation vollständig durchgeführt. Die Befehlsfolge ist in *Tabelle 20* angegeben. Sie enthält keine Besonderheiten, die noch nicht behandelt wurden.

13.11 Division

Im nächsten Beispiel werden die Zahlen a = 27 und b = 45 miteinander multipliziert:

```
a)     27
b)     45
      ----
      135
     1080
     ----
c)   1215
```

Die Division der Zahl c = 1215 durch die Zahl a = 27 ergibt den Quotienten b = 45.

```
Divisor     Dividend   Quotient
27      /   1215      \45
            108
           ----
            135
            135
           ----
              0
```

Führen wir Multiplikation und Division in „Computer-Art" aus, dann ist, wie folgt, vorzugehen:

```
                          a     b     c
a  0027                 0027/1215\0045
b  0045 x                    1080– s
   ----                      ----
   0000   p                  0135   r
   0135+  q                  0135– q
   ----                      ----
   0135   r                  0000   p
   1080+  s
   ----
c  1215
```

Aus der Division wird s ermittelt:

0040 x 0027 = 1080

und q:

0005 x 0027 = 0135

Daraus ist ersichtlich, daß die Division das Gegenteil einer Multiplikation ist, da bei beiden Rechenoperationen die Zwischenergebnisse p, q, r und s gleich sind. Untenstehend ist ein Beispiel für das Multiplizieren und Dividieren mit binären Zahlen angegeben:

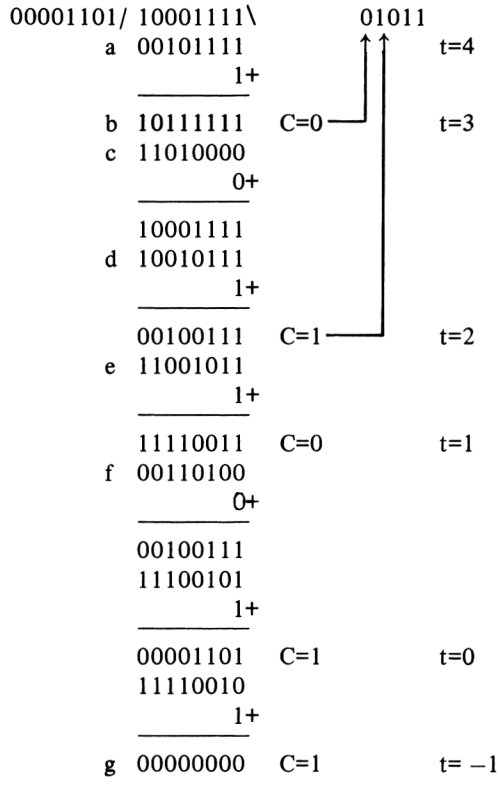
```
                          Divisor    Dividend    Quotient
00001101                  00001101/ 10001111\ 00001011
00001011x                           01101000–  u
--------                            --------
00000000    p                       00100111   t
00001101+   q                       00011010–  s
--------                            --------
00001101    r                       00001101   r
00011010+   s                       00001101–  q
--------                            --------
00100111    t                       00000000   p
01101000+   u
--------
10001111
```

Für die Division muß der Computer u.a. folgende Arbeiten ausführen:

Um u zu finden, muß der Divisor drei Stellen nach links,
um s zu finden, muß u zwei Stellen nach rechts,
um q zu finden, muß s eine Stelle nach rechts geschoben werden.

Ein solches „Rezept" muß für jeden anderen Divisor wieder anders erstellt werden und ist deshalb nicht austauschbar.

Im hier gewählten Divisonsprogramm wird der Divisor so weit als möglich nach links geschoben. Dann ist der Divisor vom Dividenden abzuziehen. Ist eine Subtraktion nicht möglich, oder besser gesagt, ist das Ergebnis negativ, dann wird eine Null im Block „Quotient" angegeben. Wird der Divisor addiert, erhält man den ursprünglichen Dividenden wieder zurück. Jetzt ist der Divisor eine Stelle nach rechts zu schieben, wieder zu subtrahieren usw. Das Arbeitsprogramm für eine Division ist im folgenden Beispiel angegeben:



a) Der Divisor befindet sich auf Speicherplatz „Divisor" und ist ganz nach links gerückt, im vorliegenden Fall um vier Stellen. Ein Zähler t wird auf 4 gesetzt. Die Subtraktion erfolgt durch Addition des Zweierkomplementes vom Divisor mit SBC-Befehl. Das Ergebnis steht auf Speicherplatz „dividiere".

b) Da C = 0, ergibt sich ein negatives Resultat. In dem Ergebnis „Quotient" wird eine „0" notiert und der Inhalt von t wird 3. Immer, wenn ein Bit in dem Ergebnis notiert wird, ist t um 1 zu dekrementieren.

c) Der Inhalt von Block „Divisor" wird zum Block „dividiert" addiert. Das Ergebnis ist der ursprüngliche Dividend.

d) Der Inhalt von „Divisor" wird eine Stelle nach rechts verschoben und vom Inhalt „Dividend" subtrahiert. Da das Ergebnis positiv ist, ist C = 1, und diese 1 wird im Ergebnis angegeben. Nach jeder Subtraktion wird der Inhalt von C im Ergebnis notiert. Der Inhalt von t wird nun 2.

e) Der Inhalt von „Divisor" wird wieder ein Bit nach rechts geschoben und subtrahiert. Das Ergebnis ist negativ, C = 0 und t wird 1.

f) Um den vorhergehenden Inhalt von Block „dividiere" zu erhalten, wird der Inhalt von Block „Divisor" wieder addiert.

g) Der Arbeitslauf setzt sich bis t = −1 fort; das Ergebnis, der Quotient, ist ermittelt und befindet sich auf Speicherplatz „Quotient".

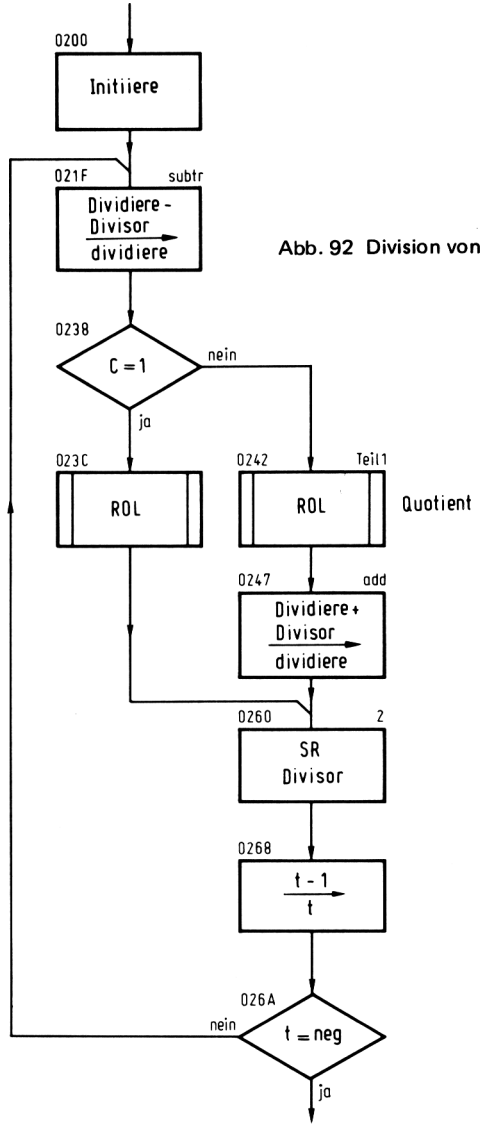


Abb. 92 Division von 32-Bit-Worten

In *Abb. 92* ist für dieses Programm das Flußdiagramm angegeben. Dabei wird von 32-Bit-Worten ausgegangen, wobei jedes Wort vier Speicherplätze einnimmt:

„Dividend“ 0000 bis 0003
„Divisor“ 0004 bis 0007
„Quotient“ 0008 bis 000B

Im Block „Init“ werden die Speicherplätze für Block „Quotient“ mit $ 00 geladen.

Tabelle 21: Divisions-Programm

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	D8	Init	CLD		Impl.
0201	18		CLC		Impl.
0202	A900		LDA		Imm. $ 00
0204	8508		STA		Z-page
0206	8509		STA		Z-page } $ 00→Quot.
0208	850A		STA		Z-page }
020A	850B		STA		Z-page
020C	AA		TAX		Impl. $ 00→x
020D	207002		JSR	SL	Abs.
0210	840C		STY	t	Z-page y→t
0212	A204		LDX		Imm. $ 04
0214	207002		JSR	SL	Abs.
0217	38		SEC		Impl.
0218	98		TYA		Impl.
0219	E50C		SBC	t	Z-page A−t
021B	850C		STA	t	Z-page A→t
021D	EAEA		NOP		ROLL
021F	38	subtr.	SEC		Impl.
0220	A500		LDA		Z-page
0222	E504		SBC		Z-page
0224	8500		STA		Z-page
0226	A501		LDA		Z-page Dividend
0228	E505		SBC		Z-page −
022A	8501		STA		Z-page Divisor
022C	A502		LDA		Z-page →
022E	E506		SBC		Z-page dividiere
0230	8502		STA		Z-page
0232	A503		LDA		Z-page
0234	E507		SBC		Z-page
0236	8503		STA		Z-page
0238	9008		BCC	Teil 1	Rel.
023A	A208		LDX		Imm. $ 08
023C	208002		JSR	ROLL	Abs. Quot.
020F	4C6002		JMP	Teil 2	Abs.

Aus Zeitersparnis beim Dividieren muß das höchstwertigste Divisorbit mit dem höchstwertigsten Dividendenbit übereinstimmen. Dazu sind beide Zahlen ganz nach links zu schieben. In *Abb. 93* ist die Subroutine vom Block „Init" angegeben.

Zum Schieben wird die Subroutine „SL" benötigt (*Abb. 94*). Diese Subroutine führt die Schiebeoperation erst bei Bedarf aus. Hierzu wird das höchste Byte vom Wortblock in den Akku geladen. Ist davon Bit 7 gleich 1, dann ist ein Schiebevorgang überflüssig, und über einen RTS-Befehl kommen wir wieder auf das Hauptprogramm zurück. Ist Bit 7 jedoch „0", dann ist ein Schiebevorgang, und zwar mit der Subroutine ROLL, erforderlich.

Fortsetzung Tabelle 21

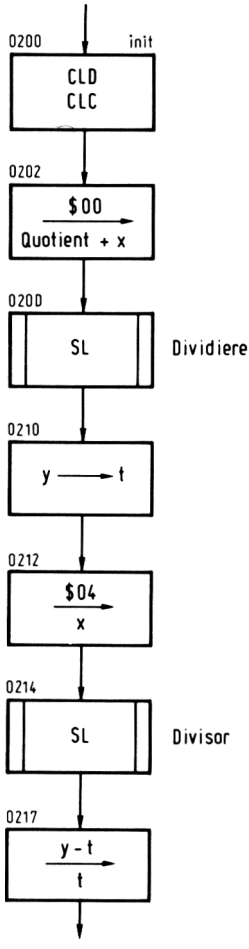
Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0242	A208	Teil 1	LDX		Imm. $ 08
0244	208002		JSR	ROLL	Abs. quot.
0247	18	ADD	CLC		Impl.
0248	A500		LDA		Z-page
024A	6504		ADC		Z-page
024C	8500		STA		Z-page
024E	A501		LDA		Z-page dividiere
0250	6505		ADC		Z-page +
0252	8501		STA		Z-page Divisor
0254	A502		LDA		Z-page →
0256	6506		ADC		Z-page dividiere
0258	8502		STA		Z-page
025A	A503		LDA		Z-page
025C	6507		ADC		Z-page
025E	8503		STA		Z-page
0260	4607	Teil 2	LSR		Z-page
0262	6606		ROR		Z-page SR
0264	6605		ROR		Z-page Divisor
0266	6604		ROR		Z-page
0268	C60C		DEC		Z-page t−1→t
026A	10B3		BPL	subtr.	Rel.
026C					

Tabelle 22: Subroutine SL

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0270	A000	SL	LDY		Imm.
0272	B503		LDA		Z-page, x
0274	4C7B02		JMP	SL 1	Abs.
0277	C8	SL 2	INY		Impl. y+1→y
0278	208002		JSR	ROLL	Abs.
027B	10FA	SL 1	BPL	SL 2	Rel.
027D	60		RTS		

Tabelle 23: Subroutine ROLL

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0280	3600	ROLL	ROL		Z-page, x
0282	3601		ROL		Z-page, x
0284	3602		ROL		Z-page, x
0286	3603		ROL		Z-page, x
0288	60		RTS		



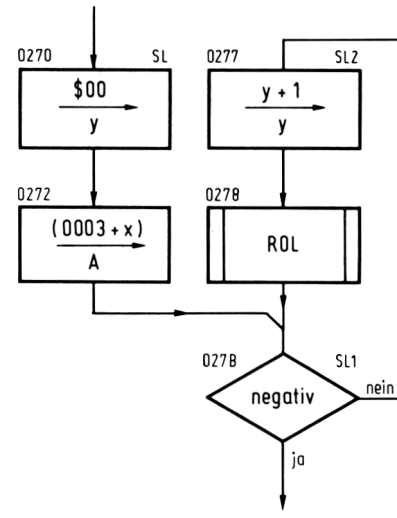


Abb. 94 Das Unterprogramm SL

Abb. 93 Unterprogramm zum Schieben der Zahlen nach links (INIT)

Die Anzahl der Wiederholungen, mit denen das Aufrücken um ein Bit des Wortblocks erfolgt, wird durch das Index-y-Register gezählt. Die Subroutine „SL" wird zweimal gebraucht, einmal für „dividiere" und einmal für „Divisor".

Nachdem der Divisor wieder nach links geschoben ist, enthält der Zähler t auf Speicherplatz 000C die Anzahl von Bits, um die der Zähler durch den Dividenden verschoben wurde. Zu diesem Zweck wird die Anzahl von Bits, um die der Dividend verschoben wurde, von der Anzahl Bits, um die der Divisor verschoben wurde, subtrahiert (y→t und y−t→t).

Nach dem Initiieren folgt die erste Subtraktion (Abb. 92). Dann wird der Inhalt von C mit der Subroutine ROLL in den Block „Quotient" geladen. Bei C = 0 wird der Inhalt vom Block „Divisor" wieder zum Inhalt vom Block „dividiere" addiert. Daraufhin wird der Inhalt vom Divisor ein Bit nach rechts verschoben, t um 1 dekrementiert und so lange t noch nicht −1 ist, wird der ganze Ablauf wiederholt.

Tabelle 21 enthält das Hauptprogramm, *Tabelle 22* die Subroutine SL und Tabelle 23

die Subroutine ROLL. Die Subroutine ROLL benutzt Zero-Page-x-Adressierung. Abhängig vom Inhalt des Index-x-Registers werden damit die Speicherplätze von „dividiere", „Divisor" oder „Quotient" erreicht. Beim Links-Schieben von „dividiere" und „Divisor" muß immer eine „0" in B0 des niedrigsten Byte eingeschoben werden, daher CLC auf Adresse 0201. Beim Schieben von „Quotient" muß der Inhalt von C nach B0 geschoben werden. Der Wert von C wird nun durch die Subtraktion:

dividiere – Divisor → dividiere

bestimmt.

14 Ein Programmbeispiel

14.1 Allgemeines

Zum Abschluß der Thematik, Mikrocomputer mit CPU 6502, folgt hier noch ein Programmbeispiel. Die im vorigen Abschnitt behandelten Programme können als Teile in anderen Programmen eingesetzt werden. Sie dienten im wesentlichen zur Vermittlung eines Einblicks in die Programmerstellung in Maschinensprache. Desweiteren konnte hierdurch die Anwendung der verschiedenen Adressierungsarten demonstriert werden.

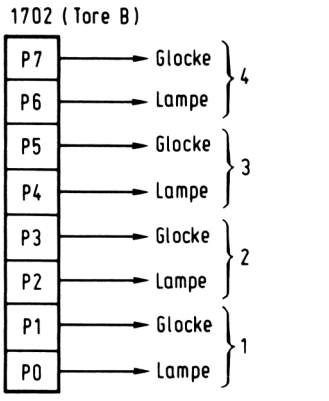
Da die Beispiele in Maschinensprache der CPU 6502 geschrieben sind, ist eine Übertragung auf andere CPU-Typen nicht ohne weiteres möglich. Das bedeutet jedoch nicht, daß beim Schreiben eines Unterprogramms für eine andere CPU nicht dem gleichen Gedankengang zur Problemlösung gefolgt werden könnte. Im einzelnen sollte jedoch den Gegebenheiten dieser CPU Rechnung getragen werden, insbesondere was Befehlssatz und mögliche Adressierungsarten betrifft.

Ein Programm für die CPU 6502 kann auch in ein Programm in Maschinensprache einer anderen CPU umgeschrieben werden. Das nun folgende Programm ist ebenfalls in der Maschinensprache der CPU 6502 geschrieben. Es handelt sich hierbei um ein Programm für eine Schaltung, die bei einem Quiz gebraucht wird.

14.2 Wer zuerst drückt . . .

Der Auftrag an den Programmierer lautet folgendermaßen: Erstelle ein Programm für einen Computer, das bei einem Quiz die Tastenstellung von max. vier Kandidaten und einem Quizmaster mit Lampen, Glocken und Summer steuert. Der Quizmaster muß zuerst das Programm mit einer Reset-Taste starten. Nach der Fragestellung drückt er eine Starttaste. Hiermit startet er die Uhr des Computers und gibt die Tasten der Kandidaten frei. Ist die Zeit für die Beantwortung verstrichen, sind diese Tasten wieder außer Betrieb, und etwa eine Sekunde lang ertönt ein Summer. Drückt einer der Kandidaten vor dem Ablauf der Zeit auf seine Taste, dann werden die Tasten der anderen Kandidaten außer Betrieb gesetzt, eine Glocke ertönt und eine Lampe leuchtet auf. Läßt der Kandidat die Taste los, verstummt die Glocke, die Lampe jedoch bleibt eingeschaltet. Jeder Kandidat hat seine eigene Lampe und seine eigene Glocke. Außerdem leuchtet auf dem Regiepult des Quizmasters eine Reset-Lampe auf, die angibt, daß zuerst das Programm mit der Reset-Taste wieder neu gestartet werden muß. Die Beantwortungszeit muß in Stufen von einer Sekunde zwischen 0 und 99 Sekunden einstellbar sein.

Am Anfang werden die Ausgangstore PB0 bis PB 7 auf Adresse 1702 für Lampen und Glocken gesetzt. P0 und P1 für Kandidat 1, P2 und P3 für Kandidat 2 usw. (*Abb. 95*). Lampen und Glocken müssen über geeignete Interface-Bausteine mit den Ausgangstoren verbunden werden. Die Tasten für die Kandidaten 1, 2, 3 und 4 sind folgerichtig an die



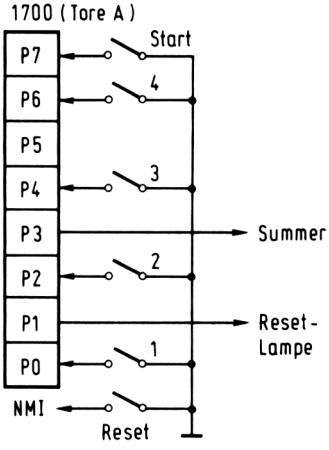


Abb. 95 Verdrahtung der I/O-Ports

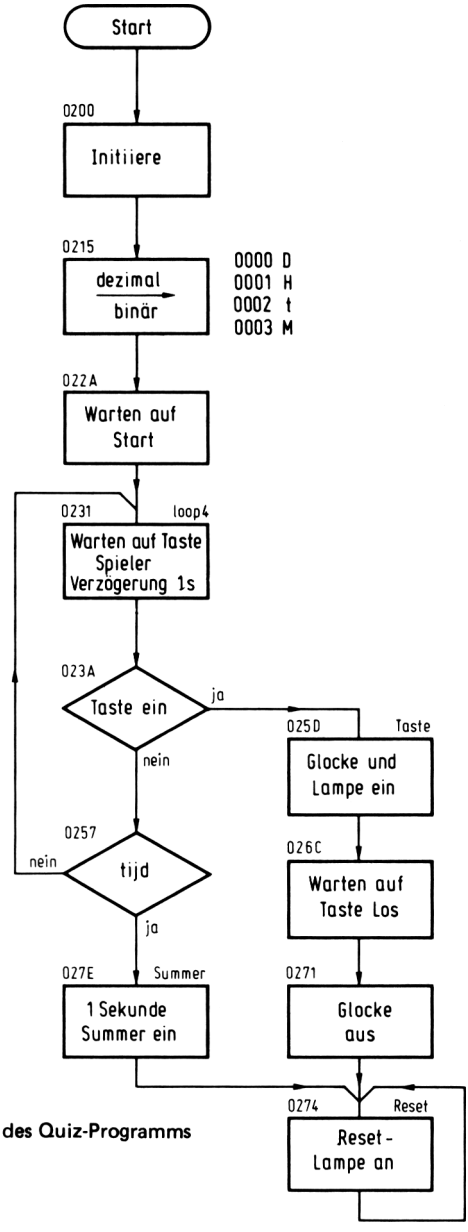


Abb. 96 Flußdiagramm des Quiz-Programms

Tore PA0, PA2, PA4 und PA6 angeschlossen, die Starttaste für den Quizmaster an PA7. Die Taste „Reset“ ist direkt mit dem $\overline{\text{NMI}}$-Eingang des Prozessors verbunden. Der Summer hat PA3 und die Reset-Lampe PA1 als Ausgangstor. Damit sind sämtliche Tasten und Schalter eingeordnet.

Vor dem Programmstart gebrauchen wir den NMI-Interrupt. Durch Drücken der Reset-Taste wird der NMI-Eingang niederohmig. Das hat zur Folge, daß die Adressenleitungen auf FFFA und später auf FFFB zu liegen kommen. Auf diese Adressen müssen wir den Vektor dieses Programmes laden, daher

FFFA: $ 00
FFFB: $ 02

Als Startadresse dieses Programmes wurde 0200 gewählt.

Das Flußdiagramm für dieses Programm geht aus *Abb. 96* hervor. Nach Initialisierung der Ein/Ausgangstore (I/O-Ports) folgt eine Dezimal-Binär-Umwandlung. Die Beantwortungszeit muß dezimal eingestellt werden können. Hierzu ist Speicherplatz 0000 (D) reserviert. Auf welche Weise dieser Speicherplatz mit der gewählten Zahl geladen werden muß, hängt vom eingesetzten Computertyp ab. Die Binär-Form der Zahl wird im Speicherplatz 0001 (H) gespeichert.

Nun muß auf das Drücken der Starttaste durch den Quizmaster gewartet werden. Ist diese Taste gedrückt, dann wird im folgenden Unterprogramm die Tastenstellung geprüft. Dieses Unterprogramm muß in einer Sekunde durchlaufen werden.

Wie oft dieses Unterprogramm durchlaufen werden muß, hängt von der eingestellten Zeit ab. Ist diese Zeit verstrichen, ohne daß ein Kandidat seine Taste gedrückt hat, dann ertönt der Summer (ungefähr eine Sekunde lang), wonach in einem Wartezyklus die „Reset"-Lampe aufleuchtet.

Wird jedoch innerhalb des Zeitintervalls eine Taste von einem Kandidaten gedrückt, dann schalten sich sowohl seine Glocke als auch Lampe ein. Wenn in einem Wartezyklus festgestellt wird, daß die Taste wieder losgelassen wurde, wird die Glocke abgeschaltet. Die Lampe dagegen leuchtet weiter. Danach wird wieder in einem Wartezyklus die „Reset"-Lampe eingeschaltet. Nur durch Bedienen der Resettaste kann dieser Zyklus eingestellt und das Programm neu gestartet werden. Der durch die „Reset"-Taste verursachte NMI-Interrupt unterbricht das Programm und bringt endlich die Adresse 0200 wieder auf die Adressenleitungen, wodurch das Programm neu anläuft.

In *Abb. 97* wird das Unterprogramm „Initialisieren" abgewickelt. Nachdem die betreffenden Tore als Ein/ oder Ausgang bestimmt sind, werden PDR A und B (Peripherie-Datenregister) mit $ 00 eingeschrieben. Sämtliche Ausgänge liegen damit auf Null. Gleichzeitig sind auch sämtliche Glocken und Lampen ausgeschaltet.

Abb. 98 stellt das Flußdiagramm für die Dezimal-Binär-Umsetzung dar. Lassen wir die einfachen Computer, bei denen die Eingabe von Daten über acht Schalter erfolgt, außer Betracht, so wird bei der Dateneingabe üblicherweise der Hexadezimal-Code eingesetzt. Führt der Anwender die Zahl 23 in Speicherplatz D (0000) ein, dann wird im Prinzip $ 23, gemeint ist aber $23_{(10)}$, eingegeben.

$\$\ 23 = 00100011_{(2)}$
$23_{(10)} = 00010111_{(2)}$

Nun gleichen die Binär-Werte der Dezimalzahlen 0 bis 9 (0000 bis 1001) den Hexadezimal-Zahlen von 0 bis 9. Die Umsetzung ist daher recht einfach. Die zweite Tetrade von D enthält die Anzahl der Zehner. Durch Addition einer entsprechenden Anzahl der Zahl $0000\ 10\ 10_{(2)} = \$\ 0A$ (00001010 + 00001010 + ...) wird der Binärwert der Zehner

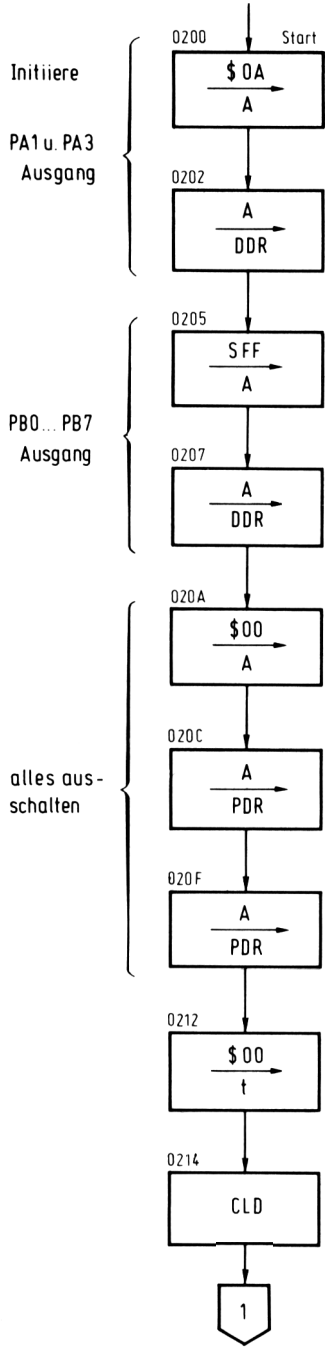


Abb. 97 Das Unterprogramm „Initialisieren"

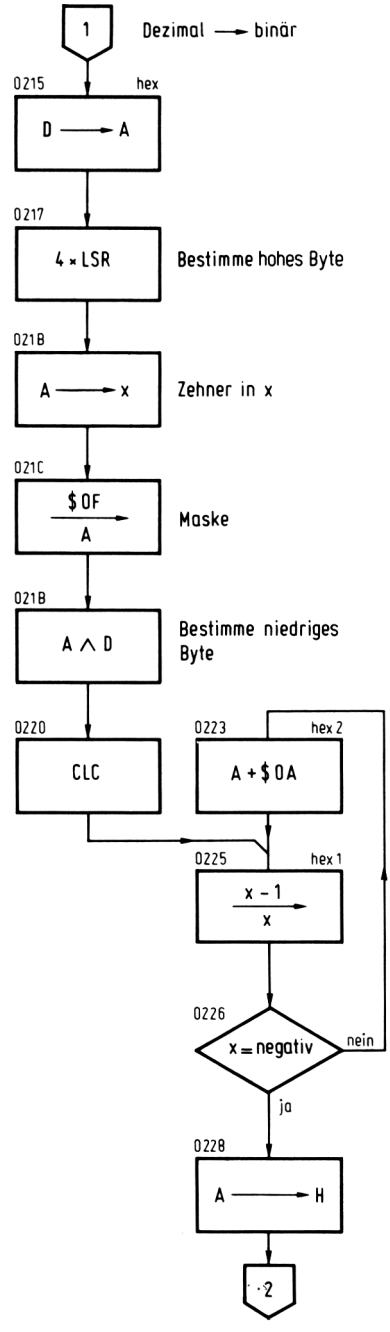


Abb. 98 Unterprogramm zur Dezimal-Binär-Umwandlung

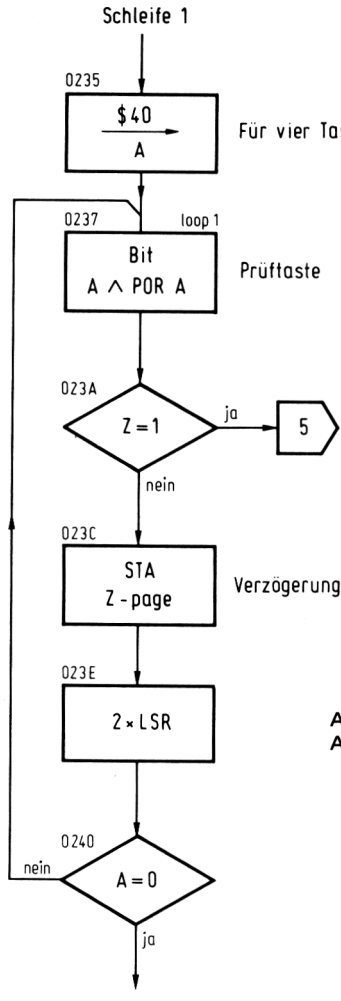


Abb. 99 Flußdiagramm zur Abfrage der Tasten

ermittelt. Dieser wird zum Binärwert der Einer (erste Tetrade von D) addiert und ergibt das gesuchte Resultat. *Abb. 98* läßt erkennen, daß durch viermaligen LSR-Befehl die hohe Tetrade der Dezimalzahl in D gefunden und wiederum durch D→A in den Akku eingegeben wird. Für die Zahl 23:

	00100011	$ 23
1.	00010001	$ 11
2.	00001000	$ 08
3.	00000100	$ 04
4.	00000010	$ 02

Dieses Ergebnis wird in das Index-x-Register eingegeben. Mit Hilfe der Maske 0001111 gelangt die niedrige Tetrade von D in den Akku, wobei durch die Schleife hex 2 x-mal die

Zahl $0A addiert wird. Darin bedeutet x den Wert der hohen Tetrade von D. Der nunmehr ermittelte Hexadezimal-Wert von D wird im Speicherplatz H (0001) gespeichert.

Der Zyklus „Warten auf Start" (Abb. 96) ist einfach und bedarf keiner weiteren Erläuterung.

Die Werte der Tasten ergeben sich aus dem Unterprogramm, dessen Flußdiagramm in *Abb. 99* wiedergegeben ist. Der Akku wird mit der Maske $ 40 geladen, worauf in der Schleife die Werte der Tasten des vierten Kandidaten bestimmt werden:

		76543210	
Taste 4 auf Tor 6 offen,	Tore A	11110101	
Tore 1 und 3 Ausgang	Maske M	01000000	
	A∧M	01000000	Z=0
		76543210	
Taste 4 auf Tor 6 geschlossen,	Tore A	10110101	
Tore 1 und 3 Ausgang	Maske M	01000000	
	A∧M	00000000	Z=1

Im Fall, daß die Taste offen ist, wird Z „0" und das Programm folgt der Schleife. Der sonst unwirksame Befehl STA verursacht eine Verzögerung, worauf der Inhalt des Akku zwei Bit nach rechts geschoben wird, um Taste 3 zu überprüfen usw. Nachdem die Lage der letzten Taste (1) festgestellt ist, wird der Akku-Inhalt durch den Schiebevorgang „0". Ein Sprung nach „loop" findet nicht mehr statt. Nun kommen wir in die zweite Schleife. Die Schleife von loop 1 erfordert 16 Taktzeiten:

	Taktzeiten		
BIT, Abs.	4		
BEQ	2	(Z=0)	
STA, Z-page	3		Verzögerung
LSR	2		
LSR	2		
BNE	3	(A≠0)	
	16		

Die Schleife wird viermal durchlaufen (vier Schalter) bei insgesamt

4 x 16 – 1 = 63

Taktzeiten. Diese Schleife (Schleife 1) ist Glied einer zweiten Schleife (Schleife 2), dessen Flußdiagramm in *Abb. 100* wiedergegeben ist. Hier gilt die nachfolgende Tabelle:

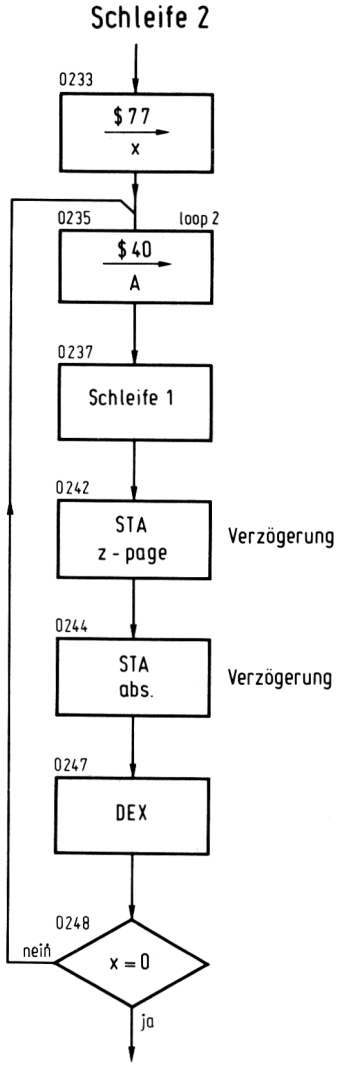


Abb. 100 Die Schleife 2 benötigt 77 Taktzyklen

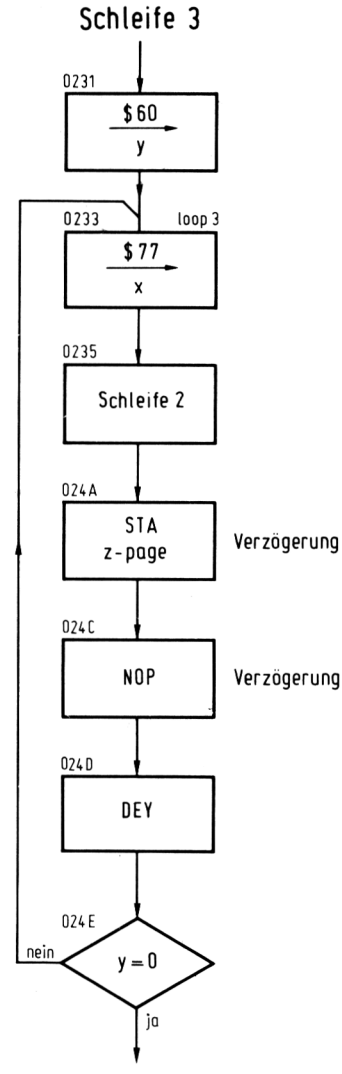


Abb. 101 Schleife 3 beansprucht 9174 Zyklen, also knapp 10 ms

	Taktzeiten	
LDA, Imm.	2	
Schleife 1	63	
STA, Abs.	4	Verzögerung
STA, Z-page	3	Verzögerung
DEX	2	
BNE	3	(x≠0)
	77	

Das Index-x-Register ist mit $ 77 = $119_{(10)}$ geladen. Die Gesamtzahl der Taktzeiten der Schleife 2 beträgt:

119 x 77 – 1 = 9162

Diese Schleife ist wieder ein Glied der Schleife 3 (*Abb. 101*)

	Taktzeiten	
LDX, Imm.	2	
Schleife 2	9162	
STA, Z-page	3	Verzögerung
NOP	2	Verzögerung
DEY	2	
BNE	3	(y≠0)
	9174	

Das Index-y-Register wird mit $ 6D = $109_{(10)}$ geladen. Die Anzahl der Taktzeiten der Schleife 3 beträgt:

109 x 9174 – 1 = 999966

Schleife 3 bildet wiederum ein Glied der Schleife 4 (*Abb. 102*). Die Gesamtzahl der Taktzeiten für das ganze Unterprogramm ergibt sich nun zu:

	Taktzeiten	
LDY, Imm.	2	
Schleife 3	999966	
INC, Z-page	5	
PHA	3	
LDA, Z-page	3	
CMP, Z-page	3	
BEQ	2	Z=0
PLA	4	
JMP	3	
	999991	

Bei einer Taktfrequenz von 1 MHz beträgt die gesamte Ablaufzeit für ein Quizspiel nicht mehr als eine Sekunde. Bei jedem Durchlauf von Schleife 4 wird der Zähler um 1 inkrementiert (am Anfang t=0). Ist der Inhalt von t gleich der eingestellten Beantwortungszeit auf Speicherplatz H (0001), wird die Schleife, um den Summer in Gang zu setzen, verlassen.

Wir kommen nun zum Unterprogramm (*Abb. 103*). Der erste Befehl lautet hier PLA. Obwohl der Akku-Inhalt, sofern dieser in den Kellerspeicher durch PHA-Befehl eingegeben ist, in der Schleife 4 nicht benötigt wird, muß durch den PLA-Befehl der Stapelzeiger wieder seinen richtigen Inhalt erhalten:

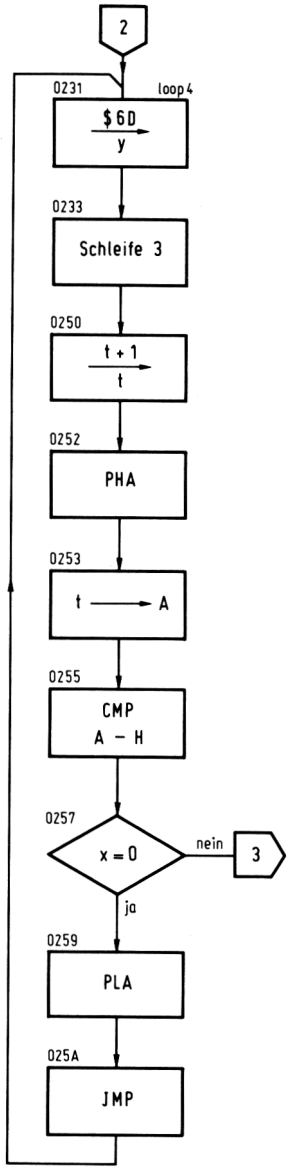


Abb. 102 Die Schleife 4 kommt schließlich auf rund eine Sekunde Laufzeit

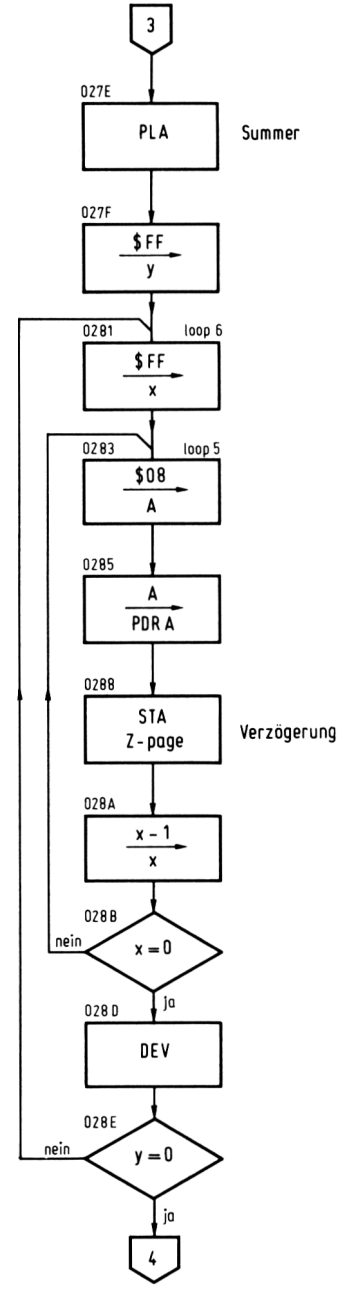


Abb. 103 Teilprogramm für die Summer-Ausgabe

Einem PHA-Befehl muß stets ein PLA-Befehl folgen!

In Schleife 5 (loop 5) wird der Summer durch Eingeben der Zahl $ 08 in PDR A eingeschaltet. P3 wird dadurch „1". Die Zahl der Taktzyklen ergibt sich zu:

	Taktzeiten	
LDA, Imm.	2	
STA, Abs.	4	
STA, Z-page	3	Verzögerung
DEX	2	
BNE	3	(x≠0)
	14	

Die Schleife wird 255 mal durchlaufen ($255_{(10)}$ = $ FF)

255 x 14 – 1 = 3569

Für die nächste Schleife (Schleife 6) beträgt die Zahl der Taktzeiten:

	Taktzeiten	
LDX, Imm.	2	
Schleife 5	3569	
DEY	2	
BNE	3	(y≠0)
	3576	

Auch diese Schleife wird 255 mal durchlaufen.

255 x 3576 – 1 = 911879

Dieses Unterprogramm dauert somit ungefähr 0,9 Sekunden, was völlig ausreichend ist für die Abgabe eines Summertones. Nach dieser Zeit erfolgt ein Sprung nach dem Unterprogramm „Reset-Lampe ein", wobei der Summer abgeschaltet wird.

Drückt einer der Kandidaten seine Taste, bevor die Zeit abgelaufen ist, dann wird über Schleife 1 nach Unterprogramm „Taste" gesprungen (*Abb. 104*). Nehmen wir an, Taste 2 wird gedrückt, dann ergibt sich ein Akkuinhalt (der Maske) von:

$ 04 = $00000100_{(2)}$

Dieser Wert geht auf Speicherplatz 0003 (N). Mittels ASL-Befehl und Addition wird die Zahl $ 0C erhalten:

Bit	76543210
M	00000100
ASL	00001000+
A	00001100

Tabelle 24. Programm: Quiz

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0200	A90A	Init	LDA		Imm. $ 0A
0202	8D0117		STA	DDR A	Abs. PA1 und PA3 Ausgang
0205	A9FF		LDA		Imm. $ FF
0207	8D0317		STA	DDR B	Abs. PB0 bis PB7 Ausgang
020A	A900		LDA		Imm. $ 00
020C	8D0017		STA	PDR A	Abs. } alles aus
020F	8D0217		STA	PDR B	Abs. } alles aus
0212	8502		STA	t	Z-page t=0
0214	D8		CLD		Impl.
0215	A500		LDA		Z-page D→A
0217	4A		LSR	accu	Z-page
0218	4A		LSR	accu	Z-page } bestimme hohe
0219	4A		LSR	accu	Z-page } Tetrade
021A	4A		LSR	accu	Z-page
021B	AA		TAX		Impl. A→x
021C	A90F		LDA		Imm. Maske
021E	2500		AND	D	Z-page niedrige Tetrade
0220	18		CLC		Impl.
0221	9002		BCC	Hex 1	Rel.
0223	690A	Hex 2	ADD		Imm. A+$ 0A } dezimal
0225	CA	Hex 1	DEX		Impl. } →
0226	10FB		BPL	Hex 2	Rel. } Hex
0228	8501		STA	H	Z-page A→H
022A	A980		LDA		Imm. $ 80
022C	2C0017	warten	BIT	PDR A	Abs. } warten auf
022F	D0FB		BNE	warten	Rel. } Starttaste ein
0231	A06D	Loop 4	LDY		Imm. $ 6D Verzögerung
0233	A277	Loop 3	LDX		Imm. $ 77 Verzögerung
0235	A940	Loop 2	LDA		Imm. $ 40 Maske
0237	2C0017	Loop 1	BIT	PDR A	Abs. Schalter ein?
023A	F021		BEQ	Taste	Rel. Taste ein
023C	8504		STA		Z-page Verzögerung
023E	4A		LSR		Accu } nächste
023F	4A		LSR		Accu } Taste
0240	D0F5		BNE	Loop 1	Rel.
0242	8504		STA		Z-page Verzögerung
0244	8D0400		STA		Abs. Verzögerung
0247	CA		DEX		Impl.
0248	D0EB		BNE	Loop 2	Rel.
024A	8504		STA		Z-page Verzögerung
024C	EA		NOP		Impl. Verzögerung
024D	88		DEY		Impl.
024E	D0E3		BNE	Loop 3	Rel.
0250	E602		INC	t	Z-page t+1→t
0252	48		PHA		Impl.

Fortsetzung Tabelle 24

Adr.	*Code*	*Label*	*Instr.*	*Operand*	*Kommentar*
0253	A502		LDA		Z-page t→A
0255	C501		CMP	H	Z-page A–H
0257	F022		BEQ	Summer	Rel.
0259	68		PLA		Impl.
025A	4C3102		JMP	Loop 4	Abs.
025D	8503	Taste	STA	M	Z-page A→M
025F	0A		ASL	accu	Impl.
0260	6503		ADC		Z-page A+M→A
0262	8D0217		STA	PDR B	Abs. Lampe und Glocke ein
0265	A503		LDA		Z-page M→A
0267	A2FF		LDX		Imm. $ FF
0269	CA	Loop 7	DEX		Impl. } Gegen Prellen
026A	D0FD		BNE	Loop 7	Rel. } Gegen Prellen
026C	2C0017	Loop 8	BIT	PDR A	Abs. } warten auf
026F	F0FB		BEQ	Loop 8	Rel. } Taste los
0271	8D0217		STA	PDR B	Abs. Lampe ein, Glocke aus
0274	68	Reset	PLA		Impl. } korrigiere
0275	68		PLA		Impl. } korrigiere
0276	68		PLA		Impl. } Stackpointer
0277	A902		LDA		Imm. $ 02
0279	8D0017	Loop 9	STA	PDR A	Abs. Resetlampe ein
027C	D0FB		BNE	Loop 9	Rel. warten auf reset
027E	68	Summer	PLA		Impl.
027F	A0FF		LDY		Imm. $ FF
0281	A2FF	Loop 6	LDX		Imm. $ FF
0283	A908	Loop 5	LDA		Imm. $ 08
0285	8D0017		STA	PDR A	Abs. Summer ein
0288	8504		STA		Z-page Verzögerung
028A	CA		DEX		Impl.
028B	D0F6		BNE	Loop 5	Rel.
028D	88		DEY		Impl.
028E	D0F1		BNE	Loop 6	Rel.
0290	F0E2		BEQ	Reset	Rel.
0292					

Dem ADC-Befehl braucht kein CLC-Befehl vorgesetzt zu werden; da durch den ASL-Befehl C schon „0" ist, wird die Zahl $ 0C nach PDR B geschrieben, dann werden Glocke und Lampe von Kandidat 2 eingeschaltet.

Der Vorgang M→A bringt die ursprüngliche Maske wieder in den Akku zurück. Danach tritt eine Verzögerung ein, um Fehler durch Kontaktprellen der Taste des Kandidaten zu vermeiden. Das folgende Unterprogramm wird, solange die Taste gedrückt bleibt, durchlaufen. Dann wird die Maske in den Akku nach PDR B geschrieben. Dadurch wird P3 „0" und die Glocke stoppt. P2 bleibt jedoch „1" und die Lampe leuchtet weiter.

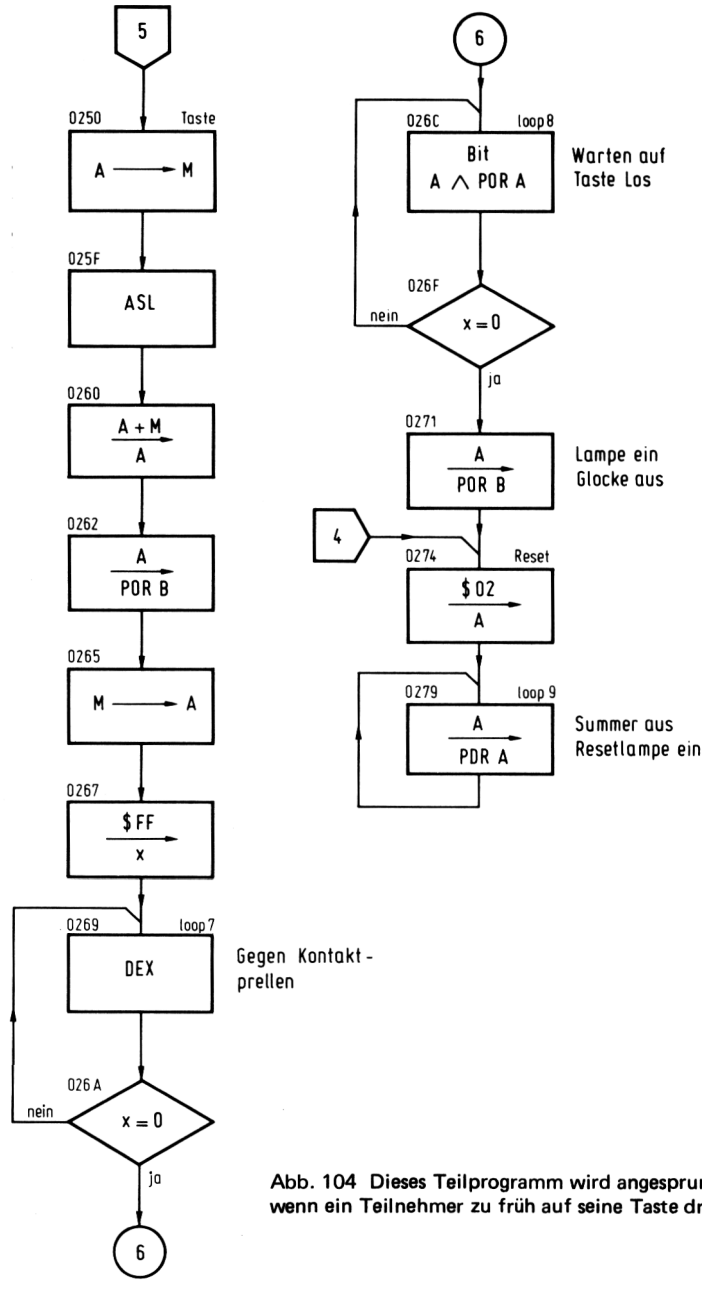


Abb. 104 Dieses Teilprogramm wird angesprungen, wenn ein Teilnehmer zu früh auf seine Taste drückt

Dadurch, daß $ 02 nach PDR A geschrieben wird, folgt das Unterprogramm, auf das die Reset-Lampe anspricht. Außerdem kommt hier der Computer in einen Wartezyklus. Durch Drücken der „Reset"-Taste muß der Quizmaster nun zuerst das Programm neu starten. Gleichzeitig erlöschen die Lampen.

In *Tabelle 24* ist die Befehlsfolge des Programms aufgeführt. Hierbei muß noch auf PLA-Befehle der Adressen 0274 bis 0276 hingewiesen werden, diese sind für die Korrektur des Stapelzeigers vorgesehen. Da das Programm mit einem NMI gestartet ist, werden beim Starten Befehlszähler und Statusregister in den Kellerspeicher eingeschrieben. Um zu vermeiden, daß durch ständiges Starten des Programms der Kellerspeicher „überläuft", muß am Programmende, wo in diesem Fall kein RTI-Befehl steht, der Stapelzeiger mit dreimaligem PLA-Befehl korrigiert werden.

15 Die CPU 6800

15.1 Allgemeines

Die CPU 6502 und die CPU 6800 sind einander verwandt, ihre Unterschiede sind deshalb nicht allzu groß. Diese zu erläutern, ist der Zweck des folgenden Abschnittes.
In *Abb. 105* sind die CPU-Register angegeben, die für das Programmieren benötigt werden. Anstelle von einem, finden wir hier zwei 8-Bit-Akku-Register A und B.
Mit einer Ausnahme sind alle Aufgaben, die an den Akku A gestellt werden, auch mit Akku B durchführbar. Dabei ergeben sich auch noch gegenseitige Möglichkeiten zwischen den beiden Akku-Inhalten.

Es ist nur ein Index-Register vorhanden, jedoch 16 Bit lang. Dieser Unterschied zur CPU 6502 beeinflußt natürlich die indizierten Adressierungsarten. Die CPU 6800 hat den gleichen Befehlszähler wie die CPU 6502. Aus diesem Grunde enthält die CPU 6800 auch einen Adressenbus mit 16 Leitungen für $2^{16} = 65536$ Speicherplätze und einen Datenbus mit acht Leitungen.

Der Stapelzeiger ist ein 16-Bit-Register. Sein Inhalt liegt nach dem Löschen der CPU nicht fest, muß aber am Programmanlauf mit einem bestimmten Befehl geladen werden. Der Stapelspeicher kann auf jede beliebige Stelle im Speicher initialisiert werden.

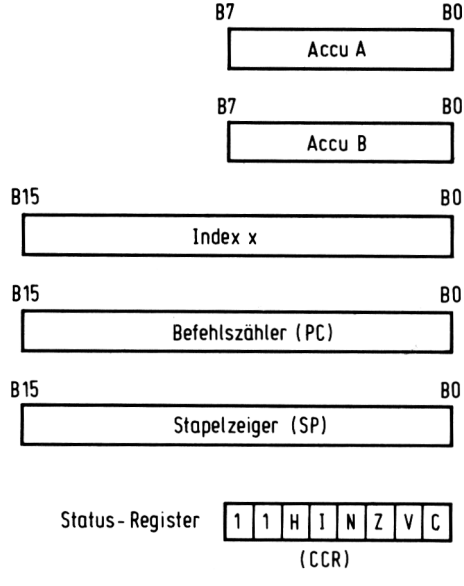


Abb. 105 Registerstruktur und Statusregister-Aufbau bei der CPU 6800

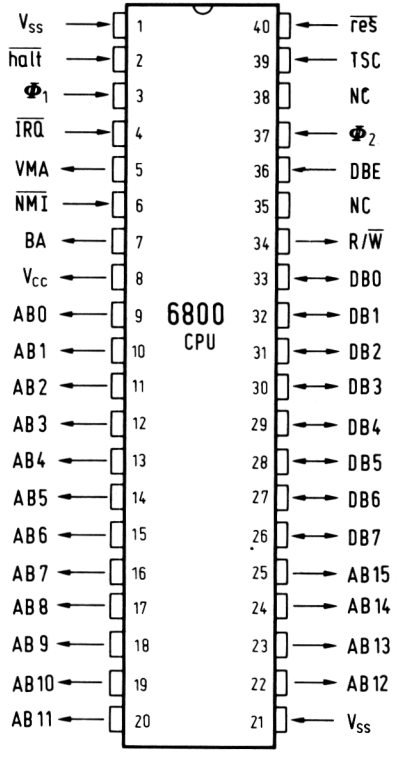


Abb. 106 Pinbelegung der CPU 6800

Das Statusregister enthält (in der CPU 6800 mit CCR bezeichnet) die Statusflags H, I, N, Z, V und C. Während die B- und D-Flags fehlen, befindet sich im Statusregister der H-Flag von „Half carry".

In *Abb. 106* sind die Anschlüsse der CPU 6800 angegeben, von denen die meisten mit der CPU 6502 übereinstimmen. Nachfolgende Unterschiede sind unten aufgeführt:

Anschluß 2: Halt. Nachdem dieser Eingang auf Pegel „L" gelegt wurde, arbeitet die CPU den im Augenblick behandelten Befehl ab und tritt dann in einen Wartezyklus. Die Ausgangspuffer liegen dann auf hochohmigen Tri-state.

Anschluß 5: VMA (valid memory address). Dieser Ausgang dient zur Kennzeichnung für das Peripheriegerät, daß eine Adresse auf dem Adressenbus zur Verfügung steht. Hierzu liegt dieser Ausgang auf H-Pegel.

Anschluß 7: BA (bus available). Wird die CPU in den Wartezustand versetzt, befinden sich die Tri-state-Ausgänge im hochohmigen Zustand, so daß der Bus für das Peripheriegerät zur Verfügung steht. Dieser Wartezustand wird dem Peripheriegerät dadurch angekündigt, daß der BA-Ausgang auf H-Pegel liegt.

Anschluß 36. DBE (data bus enable). Liegt dieser Eingang auf H-Pegel, ist der Datenbus freigegeben. Normalerweise wird der DBE-Anschluß durch das Φ_2-Taktsignal gesteuert.

Anschluß 38: Der S0-Eingang der CPU 6502 entfällt hier.

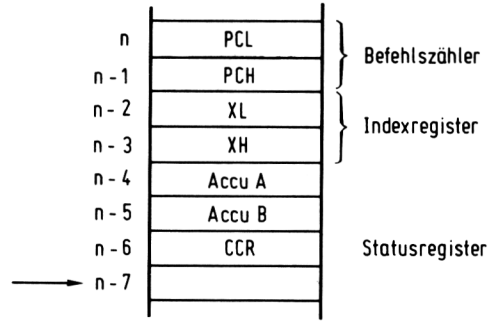


Abb. 107 So legt der 6800 seine Register auf dem Stack ab

Anschluß 39: TSC (Tristate control). Dieser Eingang hat nicht, wie der DBE-Anschluß, Einfluß auf den Datenbus, sondern setzt diesen frei, wenn der Anschluß auf L-Pegel liegt. Gleichzeitig beeinflußt er auch den Status des R/$\overline{W}$-Ausgangs.

Anschlüsse 3 und 37: Der Chip enthält keinen Taktgenerator. Ein getrennter Taktgeber muß hier die genauen Werte von Φ_1 und Φ_2 an die CPU liefern (gebräuchlich ist eine Taktfrequenz von 10^6 Hz).

Anschlüsse 4, 6 und 40: Im Prinzip stimmen diese Eingänge mit denen der CPU 6502 überein. Es wird jedoch eine andere Speicherplatzeinteilung für die Systemvektoren und Interrupt-Programme vorgenommen:

FFF8, FFF9 IRQ-Vektor,
FFFA, FFFB Software-Interrupt-Vektor,
FFFC, FFFD NMI-Vektor,
FFFE, FFFF Reset-Vektor.

Bei einem Interrupt werden mit Ausnahme des Stapelzeigers alle anderen in Abb. 105 gezeichneten Register in den Kellerspeicher geschrieben. Die Reihenfolge ist in Abb. 107 angegeben.

Der Inhalt des Stapelzeigers vor einem Interrupt wird mit n und danach mit n-7 gekennzeichnet. Dieser letzte weist dann einen leeren Speicherplatz direkt unter dem „Stapel" an.

15.2 Adressierungsarten

Die Zahl der Adressierungsarten ist wesentlich geringer als bei der CPU 6502, insbesondere durch das Fehlen des Index-y-Registers.

„Unmittelbare Adressierung". Diese Adressierungsart stimmt mit der gleichlautenden der CPU 6502 überein.

„Erweiterte Adressierung", entspricht der absoluten Adressierung der CPU 6502. Abweichend hiervon wird nach dem Operationscode zuerst der H-Byte und dann der L-Byte der Adresse angegeben. Wenn der Akku A mit $ 23, auf Speicherplatz 0313, geladen werden muß, dann lautet der Befehl B6 03 13.

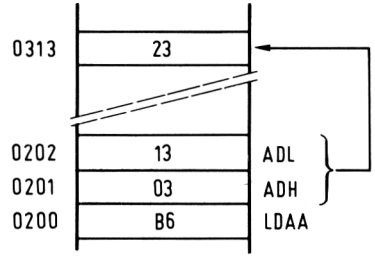


Abb. 108 Direkte Adressierung beim 6800, hier mit dem Befehl LDAA

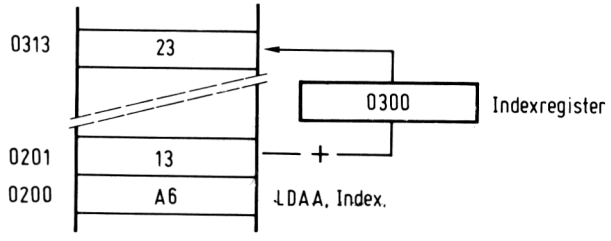


Abb. 109 Bei der indizierten Adressierung benützt der 6800 ein Byte als Argument, während das Indexregister 16 Bit lang ist

Der Operationscode von LDAA (load accu A) ist B6 (*Abb. 108*).
Die „direkte Adressierung" stimmt mit der Zero-Page-Adressierung der CPU 6502 überein.

Die „implizierte Adressierung" stimmt mit der implizierten Adressierungsart der CPU 6502 überein. Die bei der CPU 6502 mit Akku-Adressierung bezeichneten Befehle sind bei der CPU 6800 in der implizierten Adressierung enthalten.

„Relative Adressierung". Auch diese Adressierungsart entspricht ganz der gleichlautenden Adressierungsart der CPU 6502. Gilt bei der CPU 6502 diese Adressierungsart nur für bedingte Sprungbefehle, so können bei der CPU 6800 auch zwei unbedingte Sprungbefehle mit dieser Adressierungsart ausgeführt werden (branch always und branch to subroutine).

Die „indizierte Adressierung" ist ein Zwei-Byte-Befehl. Nach dem Opcode folgt eine Hexadezimalzahl, die, zum Inhalt des Index-Registers hinzuaddiert, eine Zwei-Byte-Adresse liefert. Angenommen, Akku A muß mit $ 23 auf Adresse 0313 geladen werden, das zweite Befehls-Byte ist $ 13, dann ergibt sich der Inhalt des Index-Registers $ 0300 zu:
$ 0300 + $ 13 = $ 0313 (*Abb. 109*).

15.3 Befehlssatz

Der Befehlssatz der CPU 6800 ist wesentlich umfangreicher als der der CPU 6502. Das hat seine Ursache nicht allein darin, daß hier zwei Akkus vorhanden sind, sondern daß auch eine Reihe von Befehlen zusätzlich eingebaut ist, die in der CPU 6502 nicht enthalten sind. Der Befehlssatz ist in den *Tabellen 25a bis 25d* aufgeführt.

Tabelle 25 a

Accumulator- und Speicheroperationen

		Addressing Modes															Boolean/Arithmetic Operation	Cond. Code Reg.					
		Immed			Direct			Index			Extnd			Implied				5	4	3	2	1	0
Operations	Mnemonic	OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=	(All register labels refer to contents)	H	I	N	Z	V	C
Add	ADDA	SB	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	●	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA													18	2	1	A + B → A	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	●	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				b + M + C → B	↕	●	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A∧M → A	●	●	↕	↕	R	●
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B∧M → B	●	●	↕	↕	R	●
Bit Test	BITa	85	2	2	95	3	2	A5	5	2	B5	4	3				A∧M	●	●	↕	↕	R	●
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B∧M	●	●	↕	↕	R	●
Clear	CLR							6F	7	2	7F	6	3				00 → M	●	●	R	S	R	R
	CLRA													4F	2	1	00 → A	●	●	R	S	R	R
	CLRB													5F	2	1	00 → B	●	●	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	●	●	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	●	●	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\overline{M}$ → M	●	●	↕	↕	R	S
	COMA													43	2	1	$\overline{A}$ → A	●	●	↕	↕	R	S
	COMB													53	2	1	$\overline{B}$ → B	●	●	↕	↕	R	S
Complement, 2's	NEG							60	7	2	70	6	3				00 − M → M	●	●	↕	↕	①	②
(Negate)	NEGA													40	2	1	00 − A → A	●	●	↕	↕	①	②
	NGB													50	2	1	00 − B → B	●	●	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	●	●	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	●	●	↕	↕	④	●
	DECA													4A	2	1	A − 1 → A	●	●	↕	↕	④	●
	DECB													5A	2	1	B − 1 → B	●	●	↕	↕	④	●
Exclusive OR	EORA	88	2	2	98	3	2	AB	5	2	B8	4	3				A∨M → A	●	●	↕	↕	R	●
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B∨M → B	●	●	↕	↕	R	●

Akkumulator- und Speicheroperationen

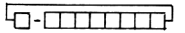
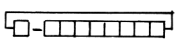
		Addressing Modes													Boolean/Arithmetic Operation	Cond. Code Reg.					
		Immed			Direct			Index			Extnd			Implied			(All register labels	5	4	3	2	1	0
Operations	Mnemonic	OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=	refer to contents)	H	I	N	Z	V	C
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	●	●	↕	↕	⑤	●
	INCA													4C	2	1	A + 1 → A	●	●	↕	↕	⑤	●
	INCB													5C	2	1	B + 1 · B	●	●	↕	↕	⑤	●
Load Acmlts	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	●	●	↕	↕	R	●
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	●	●	↕	↕	R	●
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				AVM → A	●	●	↕	↕	R	●
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				BVM → B	●	●	↕	↕	R	●
Push Data	PSHA													36	4	1	A → MSP. SP→1 → SP	●	●	●	●	●	●
	PSHB													37	4	1	B → MSP. SP→1 → SP	●	●	●	●	●	●
Pull Data	PULA													32	4	1	SP + 1 → SP. MSP → A	●	●	●	●	●	●
	PULB													33	4	1	SP + 1 → SP. MSP → B	●	●	●	●	●	●
Rotate Left	ROL							69	7	2	79	6	3				M	●	●	↕	↕	⑥	↕
	ROLA													49	2	1	A 	●	●	↕	↕	⑥	↕
	ROLB													59	2	1	B C b bO	●	●	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M	●	●	↕	↕	⑥	↕
	RORA													46	2	1	A 	●	●	↕	↕	⑥	↕
	RORB													56	2	1	B C b bO	●	●	↕	↕	⑥	↕
Shift, Left, Arithmetic	ASL							68	7	2	78	6	3				M	●	●	↕	↕	⑥	↕
	ASLA													48	2	1	A	●	●	↕	↕	⑥	↕
	ASLB													58	2	1	B C b bO	●	●	↕	↕	⑥	↕
Shift, Right, Arithmetic	ASR							67	7	2	77	6	3				M	●	●	↕	↕	⑥	↕
	ASRA													47	2	1	A	●	●	↕	↕	⑥	↕
	ASRB													57	2	1	B b bO C	●	●	↕	↕	⑥	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M	●	●	R	↕	⑥	↕
	LSRA													44	2	1	A	●	●	R	↕	⑥	↕
	LSRB													54	2	1	B b bO C	●	●	R	↕	⑥	↕
Store Acmltr.	STAA						2	A7	6	2	B7	5	3				A → M	●	●	↕	↕	R	●
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	●	●	↕	↕	R	●
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A→M → A	●	●	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B→M → B	●	●	↕	↕	↕	↕
Subtract Acmltrs.	SBA													10	2	1	A→B → A	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A→M→C → A	●	●	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B→M→C → B	●	●	↕	↕	↕	↕

Tabelle 25a (Fortsetzung)

Operations	Mnemonic	Addressing Modes: Immed OP	Immed ~	Immed =	Direct OP	Direct ~	Direct =	Index OP	Index ~	Index =	Extnd OP	Extnd ~	Extnd =	Implied OP	Implied ~	Implied =	Boolean/Arithmetic Operation (All register labels refer to contents)	Cond. Code Reg. 5 (H)	4 (I)	3 (N)	2 (Z)	1 (V)	0 (C)
Transfer Acmitrs	TAB													16	2	1	A → B	●	●	↕	↕	R	●
	TBA													17	2	1	B → A	●	●	↕	↕	R	●
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M–00	●	●	↕	↕	R	R
	TSTA													4D	2	1	A–00	●	●	↕	↕	R	R
	TSTB													5D	2	1	B–00	●	●	↕	↕	R	R
																		H	I	N	Z	V	C

Legend:

OP	Operation Code (Hexadecimal)
~	Number of MPU Cycles
#	Number of Program Bytes
+	Arithmetic Plus
–	Arithmetic Minus
A	Boolean AND
MSP	Contents of memory location pointes to be Stack Pointer
V	Boolean Inclusive OR
⊻	Boolean Exclusive OR
M	Complement of M
→	Transfer Into
0	Bit = Zero
00	Byte = Zero
C	Borrow

Note – Accumulator addressing mode instructions are included in the column for IMPLIED addressing

Condition Code Symbols

H	Half-carry from bit 3
I	Interrupt mask
N	negative (sign bit)
Z	Zero (byte)
V	Overflow, 2's complement
C	Carry from bit 7
R	Reset always
S	Set Always
↕	Test and set if true, cleared otherwise
O	Not affected

Tabelle 25 b

Index Register and Stack Pointer Instructions

		Immed			Direct			Index			Extnd			Implied			Boolean/Arithmetic	Cond. Code Reg.					
																		5	4	3	2	1	0
Pointer Operations	Mnemonic	OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	#	Operation	H	I	N	Z	V	C
Compare Index Reg.	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				XH–M, XL–(M + 1)	●	●	①	↕	②	●
Decrement Index Reg.	DEX													09	4	1	X–1 → X	●	●	●	↕	●	●
Decrement Stack Pntr	DES													34	4	1	SP–1 → SP	●	●	●	●	●	●
Increment Index Reg.	INX													08	4	1	X + 1 → X	●	●	●	↕	●	●
Increment Stack Pntr	INS													31	4	1	SP + 1 → SP	●	●	●	●	●	●
Load Index Reg.	LDX	CE	3	3	DE	4	2	BE	6	2	FE	5	3				M → XH, (M + 1)–XL	●	●	③	↕	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	3	AE	6	2	BE	5	3				M → SPH, (M + 1) → SPL	●	●	③	↕	R	●
Store Index Reg.	STX				DF	5	2	EF	7	2	FF	6	3				XH → M, XL → (M + 1)	●	●	③	↕	R	●
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				SPH → M, SPL → (M + 1)	●	●	③	↕	R	●
Indx Reg.– Stack Pntr	TXS													35	4	1	X–1 → SP	●	●	●	●	●	●
Stack Pntr – Indx Reg.	TSX													30	4	1	SP + 1 → X	●	●	●	●	●	P

① (Bit N) Test: Sign bit of most significant (MS) Byte of result = 1?

② (Bit V) Test: 2's complement overflow from subtraction of ms byte

③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Tabelle 25 c

Jump and Branch Instructions

		Relative			Index			EXTND			IMPLIED				Cond. Code Reg.					
															5	4	3	2	1	0
Operations	Mnemonic	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Branch Test	H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	●	●	●	●	●	●
Branch If Carry Clear	BCC	24	4	2										C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	25	4	2										C = 1	●	●	●	●	●	●
Branch if = Zero	BEQ	27	4	2										Z = 1	●	●	●	●	●	●
Branch if Zero	BGE	2C	4	2										N⊻V = 0	●	●	●	●	●	●
Branch if Zero	BGT	2E	4	2										ZV(N⊻V) = 0	●	●	●	●	●	●
Branch if Higher	BHI	22	4	2										CVZ = 0	●	●	●	●	●	●
Branch If Zero	BLE	2F	4	2										ZV(N⊻V) = 1	●	●	●	●	●	●
Branch If Lower or Same	BLS	23	4	2										CVZ = 1	●	●	●	●	●	●
Branch If Zero	BLT	2D	4	2										N⊻V = 1	●	●	●	●	●	●
Branch If Minus	BMI	2B	4	2										N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	28	4	2										V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	29	4	2										V = 1	●	●	●	●	●	●
Branch If Plus	BPL	2A	4	2										N = 0	●	●	●	●	●	●
Branch to Subroutine	BSR	8D	8	2											●	●	●	●	●	●
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	●	●	●	●	●	●
Jump To Subroutine	JSR				AD	8	2	BD	9	3					●	●	●	●	●	●
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	●	●	●	●	●	●
Return From Interrupt	RTI										3B	10	1		———			①	———	
Return From Subroutine	RTS										39	5	1		●	●	●	●	●	●
Software Interrupt	SWI						3F	12	1		3F	12	1		●	●	●	●	●	●
Wait for Interrupt*	WAI										3E	9	1	See Special Operations	●	②	●	●	●	●

* WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

① (All) Load Condition Code Register from Stack. (See Special Operations)

② (Bit 1) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Tabelle 25 d

Condition Code Register Instructions

		Implied				Cond. Code Reg.					
						5	4	3	2	1	0
Operations	**Mnemonic**	**OP**	**~**	**#**	**Boolean Operation**	H	I	N	Z	V	C
Clear Carry	CLC	OC	2	1	0 → C	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	R	●	●	●	●	●
Clear Overflow	CLV	0A	2	1	0 → V	●	●	●	●	R	●
Set Carry	SEC	0D	2	1	1 → C	●	●	●	●	●	S
Set Interrupt Mask	SEI	0F	2	1	I → I	●	S	●	●	●	●
Set Overflow	SEV	0B	2	1	1 → V	●	●	●	●	S	●
Acmltr A → CCR	TAP	06	2	1	A → CCR	———	———	———	①	———	———
CCR → Acmitr A	TPA	07	2	1	CCR → A	●	●	●	●	●	●

R = Reset
S = Set
● = Not affected
① (ALL) set according to the contents of Accumulator A.

In der Spalte für Statusregister bezeichnet S, daß das betreffende Bit auf 1 gesetzt, und R, daß es auf 0 rückgesetzt wird. Die sich von der CPU 6502 unterscheidenden Befehle sollten ungefähr in der Reihenfolge gemäß den Tabellen bezeichnet werden.

ADDA	Add memory to accu A	A+M→A
ADDB	Add memory to accu B	B+M→B
ABA	Add accu A to accu B	A+B→A

Bei diesen Befehlen findet eine Addition ohne Carry-Bit statt.

CLR	Clear memory	$ 00→M
CLRA	Clear accu A	$ 00→A
CLRB	Clear accu B	$ 00→B

Diese Befehle sprechen für sich selbst. Der Befehl CLR ersetzt die Kombination:

LDA, Imm. $ 00 → A
STA, Abs. A → M

CLRA ist ein Ein-Byte-Befehl und ersetzt LDA Imm. $ 00 → A.

CBA Vergleicht Akku A und Akku B A – B

Hier wird der Inhalt der Akkus miteinander verglichen. Man achte auf die Reihenfolge: A – B

COM	Complement, 1's, memory	$\overline{M} \rightarrow M$
COMA	Complement, 1's, accu A	$\overline{A} \rightarrow A$
COMB	Complement, 1's, accu B	$B \rightarrow B$

Diese Befehle bestimmen die Inversion eines Speicherplatzinhaltes, Akku A oder Akku B. Der COM-Befehl ersetzt die Kombination:

LDA, Imm. $ FF → A
EOR, Abs. A ∀ M → A
STA, Abs. A → M

Das Ergebnis dieser Kombination ist: $\overline{M} \rightarrow M$.
COMA ist ein Ein-Bit-Befehl und ersetzt EOR, Imm. A ∀ $ FF A

NEG	Complement, 2's, memory	$ 00 – M → M
NEGA	Complement, 2's, accu A	$ 00 – A → A
NEGB	Complement, 2's, accu B	$ 00 – B → B

Der negative Wert einer Zahl ($ 00–M) entspricht dem Zweierkomplement dieser Zahl. Mit diesen Befehlen ist es möglich, das Zweierkomplement eines Speicherplatzinhaltes, Akku A oder Akku B, zu bestimmen, ohne zuerst die Inversion zu ermitteln und dann 1 hinzuzuzählen. DAA Dezimal adjust.

Anstelle des SED-Befehls der CPU 6502, der alle Additionen, die diesem Befehl folgen, im BCD-Code ablaufen läßt, wird bei der CPU 6800 der DAA-Befehl eingesetzt. Dieser Befehl muß immer direkt nach jedem ADD-, ABA-, oder ADC-Befehl folgen. Wird der DAA-Befehl nicht angewandt, dann verläuft die betreffende Operation binär.

Der DAA-Befehl untersucht den Inhalt von C und H aus dem Statusregister und den Inhalt der betreffenden Tetraden und gibt in Abhängigkeit hiervon die erforderliche Korrektur. Bei einer dezimalen Subtraktion kann der Befehl nicht eingesetzt werden. Hierzu muß das Zehnerkomplement des Subtrahenden zum Minuend addiert werden (siehe Abschnitt 6.2).

Beim ASR-Befehl schieben sich alle Bits des betreffenden Registers nach rechts. Dabei gelangt B0 in C. Bit B7 behält seinen ursprünglichen Wert (*Abb. 110*).
Das besagt, daß eine negative Zahl (B7 = „1") nach dem Schieben auch noch als eine negative Zahl erkennbar ist. (B7 bleibt „1").

Dezimal	*Hex*	*Binär*
–84	AC	10101100
–42	D6	11010110

ASR Arithmetic shift right, memory
ASRA Arithmetic shift right, accu A
ASRB Arithmetic shift right, accu B

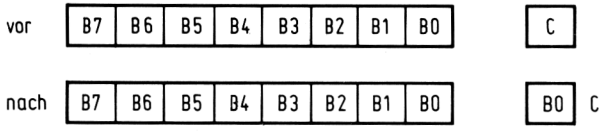


Abb. 110 Wirkung des ASR-Befehls beim 6800

Das letzte Beispiel läßt erkennen, daß die Zahl $-84_{(10)}$, deren hexadezimaler Wert AC und binärer Wert 10101100 ist, durch Division mit $2_{(10)}$ $-42_{(10)}$, Hex D6 und binär 11010110 ergeben. Auf die Binärzahl wird ein ASR-Befehl ausgeübt. Die CPU 6502 kann diesen Auftrag nur in der Befehlsfolge SEC, ROR ausführen. Dabei muß vorab bekannt sein, daß die betreffende Zahl negativ ist.

SUBA	Subtract memory from accu A	A–M →A
SUBB	Subtract memory from accu B	B–M → B
SBA	Subtract accu B from accu A	A–B → A

Bei den SUB- und SBA-Befehlen findet eine Subtraktion ohne „Borgen" statt.

TAB	Transfer accu A to accu B	A→B
TBA	Transfer accu B to accu A	B→A

Die Befehle TAB und TBA sprechen für sich selbst. Der Inhalt des Quellregisters geht dabei nicht verloren.

TST	Test memory, zero or minus	M–$ 00
TSTA	Test accu A, zero or minus	A–$ 00
TSTB	Test accu B, zero or minus	B–$ 00

In dem „Divisionsprogramm" (Abschnitt 13.11) wird die Subroutine SL erwähnt, deren Zweck es ist, den Divisor oder Dividend ganz nach links zu schieben. Zuerst muß jedoch getestet werden, ob überhaupt ein Schieben erforderlich ist. Dazu muß das Kennzeichenflag N des Statusregisters denselben Wert erhalten wie B7 des höchsten Byte vom Divisor oder Dividend. Hierzu muß mit dieser Zahl operiert werden, daher LDA, Z-page, x oder Adresse 0272 (Abb. 94, Tabelle 22).

Der ursprüngliche Akku-Inhalt geht bei dieser Operation jedoch verloren, was in bestimmten Fällen zu Schwierigkeiten führen kann. In dieser Situation sind die TST-, TSTA- oder TSTB-Befehle der CPU 6800 von besonderem Wert.

Durch Subtraktion von $ 00 der betreffenden Zahl verändert sich diese Zahl nicht mehr, N und Z aus dem Statusregister erhalten den zu dieser Zahl gehörenden Inhalt (Zahl ist Null, Z = 1; Zahl ist negativ, N = 1).

DES	decrement stack pointer	SP–1 → SP
INS	increment stack pointer	SP+1 → SP

Daß der Inhalt des Stapelzeigers noch einmal korrigiert werden muß, ergibt sich aus den Befehlen PLA auf den Adressen 0274 bis 0276 und 027E aus Tabelle 24 des Quizprogramms. Bei diesen Befehlen geht jedoch der ursprüngliche Akku-Inhalt verloren. Diese Schwierigkeit enthält der INS-Befehl der CPU 6800 nicht.

LDS	load stack pointer	M → SPH (M + 1) → SPL
STS	store stack pointer	SPH → M SPL → (M + 1)

Es ist noch anzumerken, daß zu Beginn eines jeden Programms der Stapelzeiger mit einer Adresse geladen werden muß, um die Lage des Kellerspeichers im Speicher festzulegen.

Der Stapelzeiger hat eine Länge von 16 Bit, so daß hier zwei Speicherplätze zu laden sind. Bei der CPU 6800 wird der H-Byte immer vor dem L-Byte in den Speicher eingegeben.

LDX	Load index register	M → XH (M+1) → XL
STX	Store index register	XH → M XL → (M+1)

Das Index-Register ist eine 16-Bit-Register, so daß zum Laden zwei Speicherplätze erforderlich sind. Auch hier ist die Reihenfolge „H"–„L".

BRA	Branch always
BSR	Branch to subroutine

Dies sind unbedingte Sprungbefehle, die von der relativen Adressierung Gebrauch machen. Im Gegensatz zu IMP und ISR sind es Zwei-Byte-Befehle.

BGE	Branch if greater or equal	$N \forall V = 0$
BGT	Branch if greater than	$ZV(N \forall V) = 0$
BLT	Branch if less than	$N \forall V = 1$
BLE	Branch if less or equal	$ZV(N \forall V) = 1$

Die Voraussetzungen für diese Sprungbefehle sind bereits in Abschnitt 10.4 und 10.7 besprochen und gelten sowohl für positive als auch negative Zahlen.

BHI	Branch if higher	$\overline{C} \vee Z = 0$
BLS	Branch if lower or same	$\overline{C} \vee Z = 1$

Diese Befehle erfüllen denselben Zweck wie BGT und BLE, gelten jedoch nur für positive Zahlen. Wir untersuchen nun für die drei Möglichkeiten a > M, a < M und a M den Inhalt von C und Z:

a > M a = 01101011 M = 01011001

a	01101011
$\overline{M}$	10100110
C	1+
	00010010 C=1 $\overline{C}$=0 Z=0

a < M a = 01011001 M = 01101011

a	01011001
$\overline{M}$	10010100
C	1+
	11101110 C=0 $\overline{C}$=1 Z=0

a = M a = 01101011 = M

a	01101011
$\overline{M}$	10010100
C	1+
	00000000 C=1 $\overline{C}$=0 Z=1

Aus diesen Beispielen ergibt sich, daß

für a > M: $\overline{C} \vee Z = 0$
für a ≤ M: $\overline{C} \vee Z = 1$ gilt.

NB.: $\overline{C}$ ist „geborgt".

TAP Transfer accu A to CCR; A→CCR
TPA Transfer CCR to accu A; CCR→A

Diese Transferbefehle können nur mit Akku A ausgeführt werden und ergeben sich zwangsläufig, da keine Hin- und Her-Befehle zur Verfügung stehen, die ein Schreiben nach und aus dem Kellerspeicher ermöglichen (PHP und PLP der CPU 6502). Bezüglich des TAP-Befehls ist zu vermerken, daß, unabhängig vom Akku-Inhalt von B6 und B7, die Bit von B6 und B7 des Statusregisters immer „1" werden.

16 Die CPU 8080A

16.1 Allgemeines

Die CPU 8080A weist gegenüber den Mikroprozessoren 6502 und 6800 wesentliche Unterschiede auf. Daß diese CPU auf einem anderen Konzept beruht, fällt besonders bei der Betrachtung der *Abb. 111* auf, in der die Register angegeben sind.

Der Mikroprozessor 8080A ist eine 8-Bit-CPU mit 16 Adressenleitungen für 65536 Speicherplätze. Außerdem stehen ein 8-Bit-Akku und ein 16-Bit-Befehlszähler zur Verfügung. Desweiteren enthält die 8080A-CPU einen 16-Bit-Stapelzeiger sowie ein Statusregister.

Letzteres wird hier mit „Programm-Statuswort" bezeichnet. Der Programm-Statuswort-Inhalt und die hierbei verwendeten Symbole unterscheiden sich in wesentlichen Punkten von denen der zuvor erwähnten Mikroprozessoren:

S (Sign). Das S-Bit wird nur dann gesetzt, wenn das Ergebnis einer Operation negativ ist, also immer dann, wenn Bit 7 vom Ergebnis „1" ist.

Z (Zero), Dieses Bit wird nur dann gesetzt, wenn das Ergebnis einer Operation $ 00 ist.

AC (Auxiliary Carry). Dieses Bit hat dieselbe Bedeutung wie „Half carry". Es kann in dieser CPU auch durch logische und arithmetische Funktionen beeinflußt werden.

P (Parity). Dieses Bit wird nur dann gesetzt, wenn die Anzahl von H-Bit (oder die Anzahl von Nullen) im Resultat einer Operation geradzahlig ist.

CY (Carry). Dieses Bit hat dieselbe Funktion wie C der zuvor genannten CPU's.

Weiterhin stehen dem Programmierer die Register B, C, D, E, H und L zur Verfügung. Es handelt sich um 8-Bit-Register, in denen Operanden-Daten gespeichert werden können

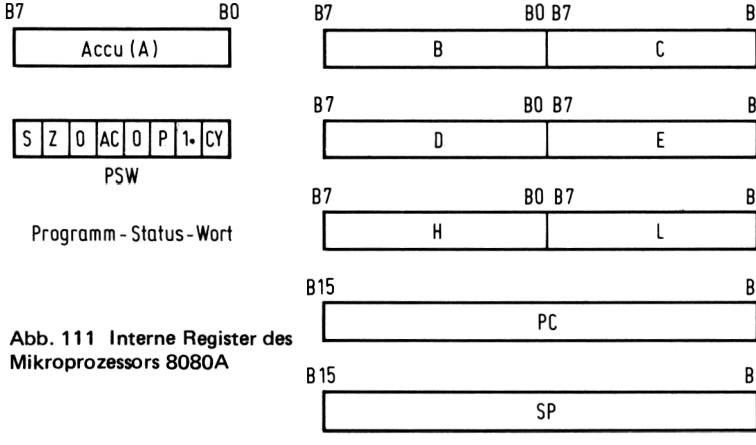


Abb. 111 Interne Register des Mikroprozessors 8080A

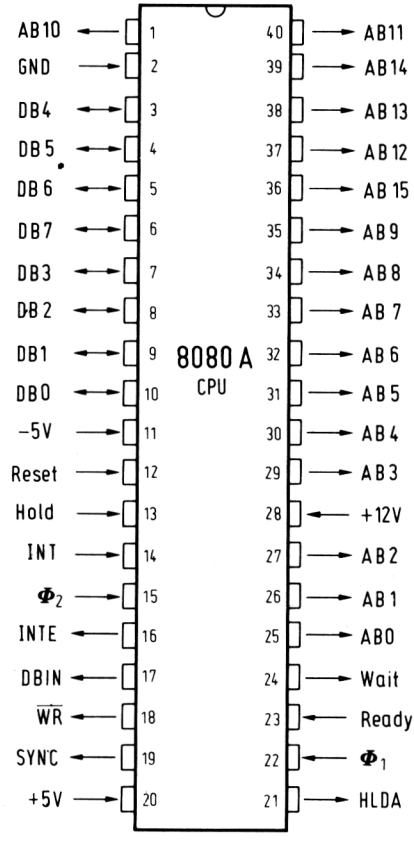


Abb. 112 Anschlußbelegung des 8080A-Chips mit drei Versorgungsspannungen

und die einen sehr schnellen und einfachen Zugriff ermöglichen. Diese Register können auch paarweise (BC, DE und HL) als Sechzehnbit-Register verwendet werden.

Die Anschlußbelegung der CPU geht aus *Abb. 112* hervor.

Die Anschlüsse 22 und 15 sind die Eingänge für ein 2-Phasen-Taktsignal (Φ_1 und Φ_2). Da der Chip keinen Taktgenerator enthält, ist bei der CPU 8080A ein externes IC mit Taktgenerator erforderlich (8224-Oszillator und Treiber). Die Signalfrequenz beträgt 2 MHz, daher enthält ein Taktzyklus eine Taktperiode von 0,5 μs.

Zur Ausführung eines Befehls sind eine Reihe von Maschinenzyklen erforderlich, wobei jeder Maschinenzyklus aus mehreren Taktperioden aufgebaut ist. Wieviele Maschinenzyklen für einen Befehl erforderlich sind und wie groß die Anzahl von Taktperioden pro Ma-

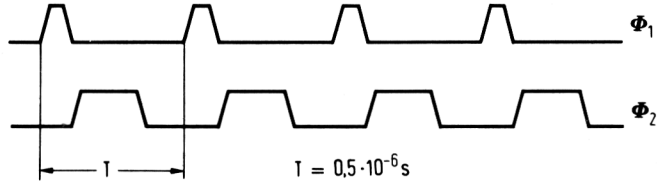


Abb. 113 Zweiphasen-Taktsignal beim 8080A

schinenzyklus ist, hängt im wesentlichen von dem betreffenden Befehl ab. Pro Befehl kann diese zwischen 1 und 5 Maschinenzyklen variieren, wogegen die Anzahl der Taktperioden pro Maschinenzyklus minimal 3 und maximal 5 betragen kann. Der erste Maschinenzyklus hat mindestens eine Länge von vier Taktperioden. Die Taktimpulse sind nicht symmetrisch. Aus *Abb. 113* ist die Form von Φ_1 und Φ_2 ersichtlich.

Am „SYNC"-Ausgang, Anschluß 19, steht ein Synchronisierungssignal zur Verfügung. Dieses markiert die erste Taktperiode eines „jeden" Maschinenzyklus.

Liegt der „HOLD"-Eingang (Anschluß 13) der CPU auf „H", wird die CPU veranlaßt, in den Haltezustand zu gehen und die Tristate-Puffer werden hochohmig.

Der „HLDA"-Ausgang (Anschluß 21) (Hold acknowledge) zeigt diesen Zustand, in dem er hochohmig wird, an.

Wird der „READY"-Eingang (Anschluß 23) „L", kommt die CPU in einen Wartezustand so lange, bis dieser Eingang „H" wird und der CPU anzeigt, daß Daten auf dem Datenbus verfügbar sind. Der READY-Anschluß wird auch zur Synchronisierung der CPU mit langsamen Speichern benötigt.

Der „WAIT"-Ausgang (Anschluß 24) auf „H" bestätigt den Wartezustand der CPU.

Der „DBIN"-Ausgang (Databus in, Anschluß 17) zeigt an, daß die CPU für die Aufnahme von Daten auf den Datenbus empfangsbereit ist. Dieser Ausgang liegt dann auf „H" und zeigt den externen Einheiten diesen Zustand der CPU an.

Der WR-Ausgang (Anschluß 18) wird für „memory write" gebraucht. Liegt dieser Ausgang auf „L", dann sind die Daten auf dem Datenbus verfügbar.

Der INT-Ausgang (Anschluß 16) zeigt den Zustand des Interrupt-Freigabe-Flipflops in der CPU an. Der Stand dieses Flipflops wird durch Freigabe-(set) und Disable-Interrupt-Befehl bestimmt. Durch einen Disable-Interrupt-Befehl wird ein auftretender Interrupt nicht akzeptiert.

Der INT-Eingang (interrupt request, Anschluß 14) verursacht, wenn er auf H liegt und das Interrupt-Freigabe-Flipflop gesetzt ist, einen Interrupt. Der laufende Befehl wird noch durch die CPU abgearbeitet.

Wir nehmen an, daß der Prozessor noch mit der Bearbeitung eines Dreibyte-Befehls, von dem der Opcode auf Adresse 0200 steht, wenn der Interrupt auftritt, beschäftigt ist (*Abb. 114*). Dieser Befehl wird abgearbeitet, danach hat der Befehlszähler den Inhalt 0203. Nun hält der Prozessor mit diesem „Hauptprogramm" an, der Befehlszähler wird nicht mehr inkrementiert. Der Freigabe-Interrupt-Flipflop geht in den Disable-Status, so daß eine folgende Interrupt-Anfrage nicht berücksichtigt wird.

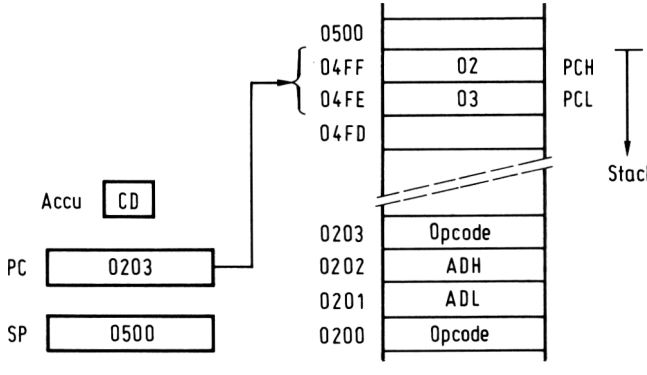


Abb. 114 Bearbeitung eines Interrupts beim 8080A

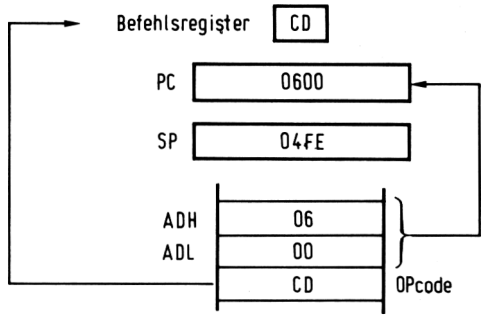


Abb. 115 Die Adresse des Interrupt-Programms muß von externen Einheiten geliefert werden

Die CPU erfordert externe Einheiten, u.a. drei Register, von denen eines den Opcode zum Aufruf einer Subroutine (CD) und zwei weitere die Adresse für das Interrupt-Programm enthält (*Abb. 115*).

Während des nächsten Maschinenzyklus wird ein Impuls an die externen Einheiten gegeben, wodurch der Opcode CD auf den Datenbus und in ein Befehlsregister eingegeben wird. Da dieser Opcode für den Aufruf einer Subroutine bestimmt ist, überschreibt der Prozessor den Befehlszählerinhalt in den Kellerspeicher (auf Adresse 04FF und 04FE, z.B. in der Reihenfolge H–L, Abb. 115). Mit den beiden nächsten Maschinenzyklen wird die Startadresse des Interrupt-Programms über den Datenbus in den Prozessor geladen, wie das auch beim Abarbeiten eines Befehls zum Abruf einer Subroutine geschieht (in diesem Beispiel Adresse 0600). Am Ende des Interrupt-Programms muß der Befehl zum Rücksprung des Programms auf das Hauptprogramm stehen. Auch dies ist derselbe Befehl wie bei einer Subroutine. Der Inhalt vom Befehlszähler wird nun wieder aus dem Kellerspeicher gelesen, und mit dem Einlesen des Opcode auf die vom Befehlszähler angewiesene Adresse wird das Hauptprogramm weiter verfolgt.

Mit Hilfe eines Signals aus der CPU und den externen Einheiten verläuft bei der CPU 8080A die Interrupt-Verarbeitung wie beim Abruf einer Subroutine.

Der Reset-Eingang (Anschluß 12) dient zum „Löschen" der CPU. Dieser Eingang muß mind. über drei Taktperioden auf „H" liegen. Das führt dazu, daß sämtliche Bits des Befehlszählers auf „0" gesetzt werden. Außerdem kommen die Unterbrechungsanfrage in den Disable-Zustand, die Logik des internen Unterbrechungs-Flipflops (INTE) und der HLDA-Ausgang in Stellung „clear".

Nun ist der Inhalt des Befehlszählers $ 0000 geworden, so daß, nachdem der Reset-Eingang auf L liegt, die CPU den Operationscode auf diese Adresse eingeben kann. Nach dem Einschalten der Spannungsversorgung wird der RESET-Eingang auch für den Systemprogrammstart benötigt.

Die Spannungsversorgung der CPU wird an die Anschlüsse 28 (+ 12V ± 5%), 20 (+ 5 V ± 5%), 11 (−5 V ± 5%) und 2 (GND) angeschlossen.

Die Anzahl der Anschlüsse der CPU reicht nicht aus, um alle Steuersignale, die das Computersystem erfordert, herauszuführen. Deshalb wird der Datenbus nicht nur für den Datenaustausch zwischen Speicher und CPU, sondern auch zur Weitergabe der CPU-Zustandsinformation zur Steuerlogik (control signal logic) eingesetzt. Während der zweiten Taktphase eines jeden Maschinenzyklus enthält der Datenbus die Daten der Zu-

standsinformation. Diese Daten betreffen den Zustand der CPU im Maschinenzyklus und können sein:

instruction fetch,
memory read,
memory write,
stack read,
stack write,
input read,
output write,
interrupt acknowledge,
halt acknowledge,
interrupt acknowledge while halt.

In welchem Status der Prozessor sich befindet, hängt von dem Befehl, der vom Prozessor ausgeführt wird, oder von dem Eintritt eines Interrupt-Befehls ab.
Als „Control-signal-Logik" kann der IC 8228 eingesetzt werden. Dieser IC bildet u.a. aus den Statusinformationen des Datenbusses folgende Signale:

INTA MEMR MEMW I/OR und I/OW.

Weiterhin enthält dieses IC die Pufferstufen für den externen Datenbus (*Abb. 116*). Der Datenbus, der in den 6800- und 6502-Systemen eine direkte Verbindung zwischen den Speichern, den Ein/Ausgangs-Toren und dem Prozessor bildet, wird hier durch den bidirektionalen Bustreiber der 8228 IC unterbrochen. Der bidirektionale Bus-Treiber (bidirektional bedeutet Datentransport in zwei Richtungen, aber nur in eine Richtung gleichzeitig), bringt den Datenbus auf einen höheren Ausgangslastfaktor, als die CPU liefern kann. Der Ausgangslastfaktor bezieht sich auf die Anzahl von Bausteinen gleichwertiger Eingangslogik, die an eine Leitung angeschlossen werden können. Mit der „Sammelschienen-Freigabe" können diese Datenbus-Treiber in den hochohmigen Tristate-Zustand gebracht werden, womit der externe Datenbus dem Peripheriegerät zur Verfügung steht.

Die Systemsteuerlogik empfängt die Daten, die den CPU-Status betreffen, über den Datenbus zwischen CPU und der Systemsteuerlogik während der zweiten Taktphase eines jeden Maschinenzyklus. Das Markieren dieser Taktphase übernimmt ein besonderer Synchronisierungsimpuls. Dieser ist ein „status strobe"-Impuls ($\overline{\text{STSTB}}$-Eingang), der vom 8224 Taktgenerator-Treiber ausgeht und eine Variante des „SYNC"-Signal der CPU 8080A darstellt. Abhängig vom Prozessor-Status wird eine der Leitungen $\overline{\text{INTA}}$, $\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, $\overline{\text{I/OR}}$ oder $\overline{\text{I/OW}}$ aktiv (L).

Wie bereits beschrieben, arbeitet der Prozessor nach einer Unterbrechungsanfrage zuerst den laufenden Befehl ab. Innerhalb des danach folgenden Maschinenzyklus und während dessen zweiter Taktphase setzt der Prozessor ein Codewort auf den Datenbus, das zum Interrupt-Status gehört. Darauf reagiert die Steuerlogik des IC 8228 und legt den INTA-Ausgang auf L.

Hierdurch wird die externe Interrupt-Logik aktiv, die, im richtigen Rhythmus durch das Signal gesteuert, zuerst den Operationscode CD, dann ADL und endlich ADH auf die Datenleitungen legt, die nun wieder durch den Prozessor freigegeben sind. Der Prozessor verarbeitet einen Interrupt wie den Abruf einer Subroutine.

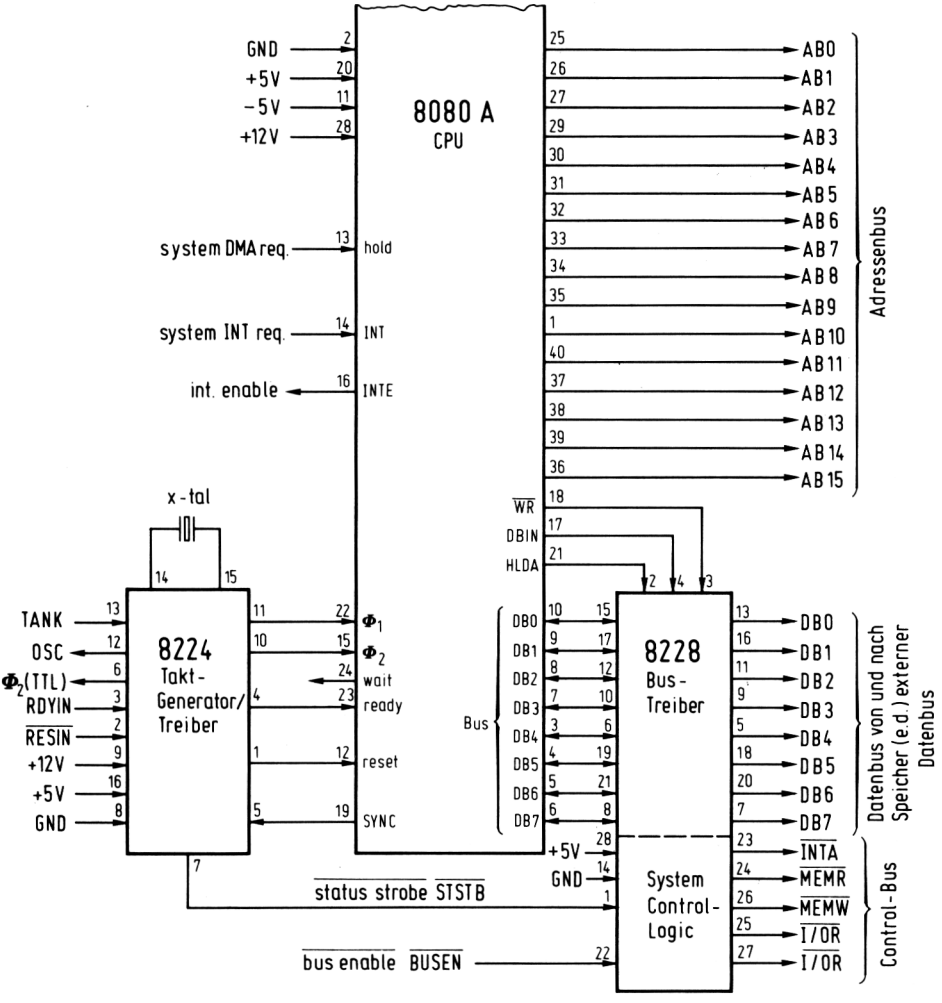


Abb. 116 Neben dem 8080A-Chip werden noch zwei weitere Bausteine zum Aufbau der vollständigen Zentraleinheit benötigt

Die weiteren Ausgangsleitungen der IC 8228 werden auf gleiche Weise aktiviert. Der Systemsteuerbaustein setzt nach Aufnahme des Codewortes während der zweiten Taktphase eines Maschinenzyklus (durch den STSTB-Impuls gekennzeichnet) die betreffende Ausgangsleitung auf L-Pegel (bei Codewort „Speicher lesen" die MEMR-, bei „Speicher einschreiben" die MEMW-Leitung usw.) Danach stehen die Datenleitungen für den Datentransfer von oder nach dem Prozessor zur Verfügung.

Im 8080-A-System sind die Ein/Ausgangstore nicht im Speicherumfang enthalten. Ein Eingangstor wird durch einen besonderen „in"-Befehl eingelesen und mit einem besonderen „out"-Befehl werden die Daten in das Ausgangstor geschrieben. Das Prinzip ist in *Abb. 117* dargestellt.

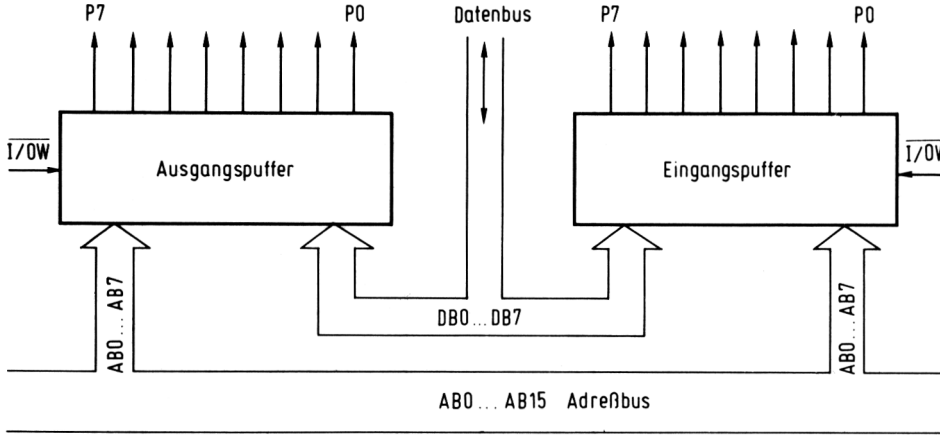


Abb. 117 I/O-Ports spricht der 8080A nicht über „memory mapping", sondern mit speziellen I/O-Befehlen an

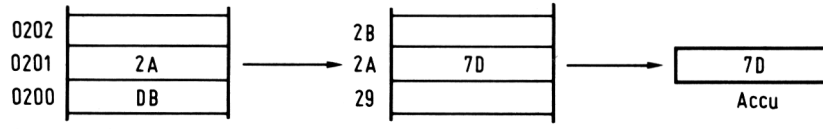


Abb. 118 Die I/O-Ports werden mit einer Ein-Byte-Adresse angesprochen

Die Ein/Ausgangs-Sammelschienen können mit den ersten acht Adressenleitungen (AB0 bis AB7) adressiert werden. Das bedeutet, daß max. $2^8 = 256$ Ein/Ausgangs-Tore mit jeweils acht Leitungen auf diese Weise adressiert werden können.

Es wird angenommen, daß im Programmablauf das Eingangstor auf Adresse 2A eingelesen werden soll. Auf Speicherplatz 0200 steht z.B. der in-Befehl mit dem Operationscode DB, auf Speicherplatz 0201 die Adresse 2A (*Abb. 118*).

Nach dem Codieren des Operationscode ergibt sich der Inhalt der Adressenleitungen auf

xxxxxxxx00101010 x = don't care

Der System-Steuerbaustein erhält dann über den Datenbus die Statusinformation „Eingang lesen", worauf die Datenleitungen frei gegeben werden. Der System-Steuerbaustein setzt dann die I/OR-Leitung auf L-Pegel, wodurch der Inhalt des Eingangstores (hier 7D) auf den Datenbus gelangt und vom Akku aufgenommen werden kann.

Das „Ausgang einschreiben" verläuft entsprechend. Jetzt wird der $\overline{\text{I/OW}}$-Ausgang auf L-Pegel gesetzt. Durch die Anpassung der $\overline{\text{I/OR}}$- und $\overline{\text{I/OW}}$-Ausgänge kann der gesamte Speicherumfang als Daten-Programm- und System-Speicher eingesetzt werden, ohne daß innerhalb dieses Speichers Platz für die Ein/Ausgangstore freigehalten werden muß.

16.2 Adressierungsarten

Die CPU 8080A besitzt Ein-, Zwei- und Drei-Byte-Befehle. Die Ein-Byte-Befehle beziehen sich vor allem auf die internen Register der CPU. Der Operationscode wird nicht hexadezimal, sondern binär eingegeben. Das liegt daran, daß der Anwender in bestimmten Fällen selbst den Code ermitteln muß, um durch gewisse Bit-Kombinationen die Wortadresse zu ersetzen, die das Ziel-(destination register) oder Quell-Register (source register) angibt. Die Register sind daher mit einer Dreibit-Zahlenkombination in der CPU angegeben:

000 B-register
001 C-register
010 D-register
011 E-register
100 H-register
101 L-register
110 M, adressiert durch HL
111 Akku

Im Operationscode wird zuerst das Zielregister und danach das Quellregister genannt. Deswegen wird die Befehlsausführung immer umgekehrt angegeben im Vergleich zu den Prozessoren 6800 und 6502.

(A) ← (A) + 1
In Worten: A wird A plus 1.

Die Klammern geben an, daß die Berechnung mit dem Akku-Inhalt ausgeführt wird. Als Beispiel der Transferbefehl:

MOV r1, r2 Move registers (r1) ← (r2)

Als Operationscode für diesen Befehl wird binär

01DDDSSS

angegeben. Durch Ersatz von DDD durch die Bitkombination des Zielregisters und SSS durch die Bitkombination des Quellregisters ergibt sich der Opcode. Für die Transferoperation ist dann der Opcode:

(E) ← (C): 0 1 | 0 1 1 | | 0 0 1 |
E C

Auch der nächste Befehl kann auf diese Weise zusammengestellt werden:

Opcode: 0 1 | 0 1 1 | | 1 1 0 |
E M

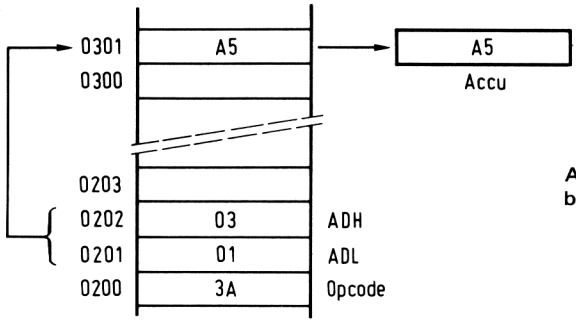


Abb. 119 Direkte Adressierung beim 8080A

Die Operation (E) ← [(H) (L)] besagt:
der Inhalt von Register E entspricht dem Inhalt des Registers, das durch den Inhalt von H und L adressiert ist. Daher die doppelten Klammern bei [(H) und (L)]. Dieser Einbyte-Befehl gehört zu der implizierten Adressierungsart.

Die unmittelbare Adressierung benutzt Zweibyte-Befehle.

Dem Operationscode folgt hierbei direkt die Zahl, mit der die Operation ausgeführt werden soll. Ein Beispiel ist der Befehl:

MVI M Move to memory immediate [(H) (L)] ← (byte 2)

Die hier ausgeführte Operation lautet:

der Speicherplatzinhalt, durch den Inhalt von H und L adressiert, ist gleich dem Inhalt des zweiten Byte. Bei der „direkten Adressierung" folgt dem Operationscode eine Zweibyte-Adresse des Operanden. Dabei steht im zweiten Byte ADL und im dritten Byte ADH. Beispiel:

LDA Load accu direct (A) ← [(byte 3) (byte 2)]

Der Akku-Inhalt entspricht dem Speicherplatzinhalt, adressiert durch den Inhalt von Byte 3 und 2.

In *Abb. 119* steht der Operationscode für LDA direkt auf Adresse 0200 und der Operand (A5) auf Adresse 0301. Der Befehl in Maschinensprache lautet nun:

3A0103

Bei der „indirekten Adressierung" werden die CPU-Register eingesetzt. Der Inhalt eines Registerpaares (B und C oder D und E) bildet die Operandenadresse:

LDAX B Load accu indirect (A) ← [(B) (C)]

Der Akku-Inhalt entspricht dem Speicherplatzinhalt, der durch den Registerinhalt B und C adressiert ist. Die Befehle dieser Adressierungsart sind ebenfalls Einbyte-Befehle. Man beachte dabei, daß die Reihenfolge ADH und ADL in den Registerpaaren mit H und L übereinstimmen.

Tabelle 26: Befehlssatz des 8080 A

Mnemonic		Binary code bits affected	Condition bytes	Number of mach. states (T)	Description of instructions	Operation
Transfer instructions						
a) Register – register						
MOV	r1, r2,	01dddsss	– – – – – 1	5	Move register to register	(r1) ← (r2)
XCHG		11101011	– – – – – 1	4	Exchange D&E, H&L	(H) ↔ (D) (L) ↔ (E)
XTHL		111000111	– – – – – 1	18	Exchange top of stack H&L	(L) ↔ ((SP)) (H) ↔ ((SP) + 1)
SPHL		11111001	– – – – – 1	5	H&L to stack pointer	(SP ← (H)(L)
Instruction Set 8080 A						
b) Memory peripheral device → register						
MOV	r, M	01ddd110	– – – – – 1	7	Move memory to register	(r) ← ((H)(L))
LDA	adr	00111010	– – – – – 3	13	Load accumulator direct	(A) ← ((byte 3) (byte 2))
LDAX	rp	00rr1010	– – – – – 1	7	Load accumulator indirect	(A) ← ((rp))
LHLD	adr	00101010	– – – – – 3	16	Load H&L direct	(L) ← ((byte 3) (byte 2)) (H) ← ((byte 3) (byte 2) + 1)
POP	rp PSW	11rr0001 11110001	– – – – – 1 Z,S,P,CY,AC	10	Pop register pair rp off stack Pop Accumulator and Flags off Stack	(rp) ← ((SP), (SP) + 1) (SP) ← (SP) + 2
IN	No	11011011	– – – – – 2	10	Input	(A) ← (data)
c) Data → register pair						
LXI	rp, adr	00rr0001	– – – – – 3	10	Load register pair immediate	(rp) → (byte 3, byte 2)
d) Register → memory, peripheral device						
MOV	M,r	01110sss	– – – – – 1	7	Move register to memory	((H)(L)) ← (r)
STA	adr	00110010	– – – – – 3	13	Store accumulator direct	((byte 3) (byte 2)) ← (A)
STAX	rp	00rr0010	– – – – – 1	7	Store accumulator indirect	((rp)) ← (A)
SHLD	adr	00100010	– – – – – 3	16	Store H&L direct	(byte 3) (byte 2)) ← (L) ((byte 3) (byte 2) + 1) ← (H)
PUSH	rp PSW	11rr0101 11110101	– – – – – 1 – – – – –	11	Push register pair rp on stack Push Accumulator and Flags on Stack	(SP)–1, (SP)–2) ← (rp) (SP) ← (SP)–2
OUT	No.	11010011	– – – – – 2	10	Output	(data) ← (A)
e) Data → register memory						
MVI	M, data	00110110	– – – – – 2	10	Move to memory immediate	((H)(L)) ← (byte 2)
MVI	r, data	00ddd110	– – – – – 2	7	Move immediate Register	(r) ← (byte 2)
Arithmetic operations						
INR	r	00ddd100	Z,S,P,–AC 1	5	Increment register	(r) ← (r) + 1
INR	M	00110100	Z,S,P,–AC 1	10	Increment memory	((H)(L)) ← ((H)(L)) + 1
DCR	r	00ddd101	Z,S,P,–AC 1	5	Decrement register	(r) ← (r)–1
DCR	M	00110101	Z,S,P,–AC 1	10	Decrement memory	((H)(L)) ← ((H))(L))–1
INX	rp	00rr0011	– – – – – 1	5	Increment register pair	(rp) ← (rp) + 1
DCX	rp	00rr1011	– – – – – 1	5	Decrement register pair	(rp) ← (rp)–1
ADD	r	10000sss	Z,S,P,CY,AC 1	4	Add register to accumulator	(A) ← (A) + (r)
ADD	M	10000110	Z,S,P,CY,AC 1	7	Add memory to accumulator	(A) ← (A) + ((H)(L))
ADC	r	10001sss	Z,S,P,CY,AC 1	4	Add register to accumulator with carry	(A) ← (A) + (r) + (CY)
ADC	M	10001110	Z,S,P,CY,AC 1	7	Add memory to accumulator with carry	(A) ← (A) + ((H)(L)) + (CY)
DAD	rp	00rr1001	– – – CY – 1	10	Add register pair to H and L	(H)(L) ← (H)(L) + (rp)
SUB	r	10010sss	Z,S,P,CY,AC 1	4	Subtract register from accumulator	(A) ← (A)–(r)

Tabelle 26 (Fortsetzung)

Mnemonic		Binary code bits affected	Condition	bytes	Number of mach. states (T)	Description of instructions	Operation
SUB	M	10010110	Z,S,P,CY,AC	1	7	Subtract memory from accumulator	(A) ← (A)−((H)(L))
SBB	r	10011sss	Z,S,P,CY,AC	1	4	Subtract register from accumulator with borrow	(A) ← (A)−(r)−($\overline{CY}$)
SBB	M	10011110	Z,S,P,CY,AC	1	7	Subtract memory from accumulator with borrow	(A) ← (A)−((H)(L))−($\overline{CY}$)
ADI	data	1100010	Z,S,P,CY,AC	2	7	Add immediate to accumulator	(A) ← (A) + (byte 2)
ACI	data	11001110	Z,S,P,CY,AC	2	7	Add immediate to accumulator with carry	(A) ← (A) + + (byte 2) + ($\overline{CY}$)
SUI	data	11010110	Z,S,P,CY,AC	2	7	Subtract immediate from accumulator	(A) ← (A)−(byte 2)
SBI	data	11011110	Z,S,P,CY,AC	2	7	Subtract immediate from accumulator with borrow	(A) ← (A)−(byte 2)−($\overline{CY}$)
DAA		00100111	Z,S,P,CY,AC	1	4	Decimal adjust accumulator	
Logic operations							
CMA		00101111	– – – – –	1	4	Complement accumulator	(A) ← ($\overline{A}$)
ANA	r	10100sss	Z,S,P,CY,AC	1	4	And register with accumulator	(A) ← (A)∧(r)
ANA	M	10100110	Z,S,P,CY,AC	1	7	And memory with accumulator	(A) ← (A)∧((H)(L))
ANI	data	11100110	Z,S,P,CY,AC	2	7	And immediate with accumulator	(A) ← (A)∧(byte 2)
ORA	r	10110sss	Z,S,P,CY,AC	1	4	Or register with accumulator	(A) ← (A)∨(r)
ORA	M	10110110	Z,S,P,CY,AC	1	7	Or memory with accumulator	(A) ← (A)∨((H)(L))
ORI	data	11110110	Z,S,P,CY,AC	2	7	Or immediate with accumulator	(A) ← (A)∨(byte 2)
XRA	r	10101sss	Z,S,P,CY,AC	1	4	Exclusive Or register with accumulator	(A) ← (A)∀(r)
XRA	M	10101110	Z,S,P,CY,AC	1	7	Exclusive Or memory with accumulator	(A) ← (A)∀((H)(L))
XRI	data	11101110	Z,S,P,CY,AC	2	7	Exclusive Or immediate with accumulator	(A) ← (A)∀(byte 2)
CMP	r	10111sss	Z,S,P,CY,AC	1	4	Compare register with accumulator	(A)−(r)
CMP	M	10111110	Z,S,P,CY,AC	1	7	Compare memory with accumulator	(A)−((H)(L))
CPI	data	11111110	Z,S,P,CY,AC	2	7	Compare immediate with with accumulator	(A)−(byte 2)
Register instructions							
a) Rotate accumulator							
RLC		00000111	– – – CY –	1	4	Rotate accumulator left	
RRC		00001111	– – – CY –	1	4	Rotate accumulator right	
RAL		00010111	– – – CY –	1	4	Rotate accumulator left through carry	
RAR		00011111	– – – CY –	1	4	Rotate accumulator right through carry	
b) Carry bit instructions							
CMC		00111111	– – – CY –	1	4	Complement carry	(CY) ← ($\overline{CY}$)
STC		00110111	– – – CY –	1	4	Set carry	($\overline{CY}$) ← 1
Jump instructions							
a) Unconditional jumps							
PCHL		11101001	– – – – –	1	5	H&L to program counter	(PCH) ← (H) (PCL) ← (L)
JMP	adr	11000011	– – – – –	3	10	Jump unconditional	(PC) ← (byte 3) (byte 2)

Tabelle 26 (Fortsetzung)

Mnemonic		Binary code	Condition bits affected	bytes	Number of mach. states (T)	Description of instructions	Operation
b) Conditional jumps							
JC	adr	11011010	- - - - -	3	10	Jump on carry	(PC) ← (byte 3) (byte 2)
JNC	adr	11010010	- - - - -	3	10	Kump on no carry	(PC) ← (byte 3) (byte 2)
JZ	adr	11001010	- - - - -	3	10	Jump on zero	(PC) ← (byte 3) (byte 2)
JNZ	adr	11000010	- - - - -	3	10	Jump on not zero	(PC) ← (byte 3) (byte 2)
JM	adr	11111010	- - - - -	3	10	Jump on minus	(PC) ← (byte 3) (byte 2)
JP	adr	11110010	- - - - -	3	10	Jump on positive	(PC) ← (byte 3) (byte 2)
JPE	adr	11101010	- - - - -	3	10	Jump on parity even	(PC) ← (byte 3) (byte 2)
JPO	adr	11100010	- - - - -	3	10	Jump on parity odd	(PC) ← (byte 3) (byte 2)
Subroutine handling							
a) Program calls							
A return address is pushed onto the stack for use by the call instructions							
CALL	adr	11001101	- - - - -	3	17	Call unconditional	(PC) ← (byte 3) (byte 2)
CC	adr	11011100	- - - - -	3	11/17	Call on carry	(PC) ← (byte 3) (byte 2)
CNC	adr	11010100	- - - - -	3	11/17	Call on no carry	(PC) ← (byte 3) (byte 2)
CZ	adr	11001100	- - - - -	3	11/17	Call on zero	(PC) ← (byte 3) (byte 2)
CNZ	adr	11000100	- - - - -	3	11/17	Call on not zero	(PC) ← (byte 3) (byte 2)
CM	adr	11111100	- - - - -	3	11/17	Call on minus	(PC) ← (byte 3) (byte 2)
CP	adr	11110100	- - - - -	3	11/17	Call on positive	(PC) ← (byte 3) (byte 2)
CPE	adr	11101100	- - - - -	3	11/17	Call on parity even	(PC) ← (byte 3) (byte 2)
CPO	adr	11100100	- - - - -	3	11/17	Call on parity odd	(PB) ← (byte 3) (byte 2)
RST	data	11NNN111	- - - - -	1	11	Restart	(PC) ← 8 · (NNN)
b) Return instructions							
RET		11001001	- - - - -	1	10	Return	
RC		11011000	- - - - -	1	5/11	Return on carry	
RNC		11010000	- - - - -	1	5/11	Return on no carry	
RZ		11001000	- - - - -	1	5/11	Return on zero	
RNZ		11000000	- - - - -	1	5/11	Return on not zero	
RM		11111000	- - - - -	1	5/11	Return on minus	
RP		11110000	- - - - -	1	5/11	Return on positive	
RPE		11101000	- - - - -	1	5/11	Return on parity even	
RPO		11100000	- - - - -	1	511	Return on parity odd	
Program interrupts							
EI		11111011	- - - - -	1	4	Enable interrupts	
DI		11110011	- - - - -	1	4	Disable interrupts	
Miscellaneous instructions							
HLT		01110110	- - - - -	1	7	Halt	
NOP		00000000	- - - - -	1	4	No operation	

Codierung Register (r):

B 000
C 001
D 010
E 011
H 100
L 101
M 110
A 111

Codierung Registerpaar (rp):

B und C 00
D und E 01
H und L 10
SP 11

16.3 Befehlssatz

Die Operationscodes sind in *Tabelle 26* für die Befehle aufgelistet, von denen viele keiner weiteren Erläuterung bedürfen, da deren Wirkungsweise aus der Spalte „Operation" hervorgeht.

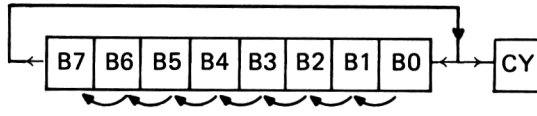
Zu bestimmten Befehlen oder Befehlsgruppen soll aber noch eine erklärende Anmerkung gegeben werden. In der Tabelle für den Befehlssatz ist u.a. die erforderliche Anzahl von Taktphasen für einen bestimmten Befehl angegeben. Die Zahl steht in der Spalte „Number of States" (T) (Schrittzahl). Aus dieser ist die Zeitdauer für jeden Befehl zu berechnen, da jeder Schritt 0,5 µs dauert.

Kein einziger Transferbefehl beeinflußt das Prozessorstatuswort. Wenn der Status einer Registerzahl bestimmt werden soll (S, Z und/oder P), dann ist ein Transferbefehl hierzu nicht geeignet; vielmehr kann z.B. die Zahl $ 00 addiert werden, wodurch sich der Registerinhalt nicht ändert, sobald ein Conditional-Flag gesetzt wurde.

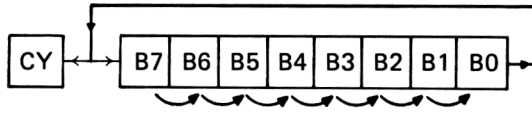
Bei den arithmetischen Operationen gibt es Befehle, die nur einige bzw. überhaupt keine Status-Flags beeinflussen.

Soll ein bedingter Sprung in Abhängigkeit vom Ergebnis einer arithmetischen Operation durchgeführt werden, dann muß bei der Befehlsauswahl auf die Beeinflussung des richtigen Kennzeichens geachtet werden. Das gilt besonders für die Dekrementier-/Inkrementierbefehle. Diese werden oft bei Sprungoperationen in Abhängigkeit vom Zählerinhalt gebraucht.

Von den Rotierbefehlen sind die RAL- und RAR-Befehle identisch mit den ROL- und ROR-Befehlen der CPU 6502. Der RLC-Befehl ruft ein Rotieren, ohne das Carry-Bit in den Rotiervorgang einzubeziehen, hervor:



Beim Rotieren erhält CY den Wert von B7, ebenso B0. Die anderen Bits wandern insgesamt eine Stelle nach links. Der RRC-Befehl führt die gleiche Operation aus, jedoch in entgegengesetzter Richtung:



Die Rotierbefehle beziehen sich ausschließlich auf den Akku-Inhalt.

Der Befehl „clear carry" ist hier nicht vorhanden. Wenn erforderlich, können gegebenenfalls die Befehle STC (set carry) und CMC (complement carry) in dieser Reihenfolge nacheinander eingegeben werden, sofern im voraus der Wert des Carry-Bit nicht bekannt ist. Ist jedoch vorab sicher, daß das Carry-Bit den Wert „1" hat, dann kann man sich auf den CMC-Befehl beschränken.

Bei den Jump-Befehlen sind die Call-Befehle für den Aufruf einer Subroutine vorgesehen. Der Sprung nach der Subroutine kann unbedingt erfolgen (call), wie der Befehl JSR der CPU 6502, aber auch bedingt (call on carry, call on minus usw.). Mit den Call-Befehlen wird der Befehlszähler in den Kellerspeicher geschrieben, damit die Adresse, bei

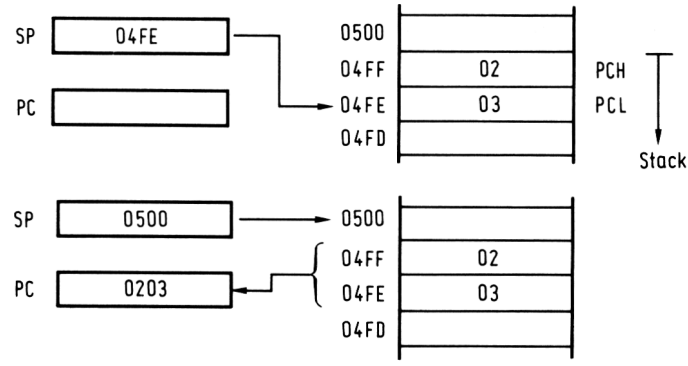


Abb. 120 Stackpointer vor einem Unterprogramm-Rücksprung

Abb. 121 Stackpointer nach dem Rücksprungbefehl

der das Hauptprogramm verlassen wurde, nicht verlorengeht. Die Arbeitsweise des Kellerspeichers unterscheidet sich nicht von der des Stack in der CPU 6800, es sei denn, der Stapelzeiger weist nicht den nächstleeren Speicherplatz, sondern den zuletzt eingeladenen an. Beim Einschreiben in den Kellerspeicher wird zunächst der Stapelzeiger um „1" dekrementiert und dann PCH in die Adresse geschrieben, die der Stapelzeiger angibt. Dann wird PCL in den Kellerspeicher auf den direkt darunter liegenden Speicherplatz geschrieben. Siehe hierzu auch Beispiel in Abb. 113.

Mit einem Rücksprungbefehl wird von der Subroutine auf das Hauptprogramm zurückgesprungen. Das geschieht sowohl unbedingt (RET, return) als auch bedingt (RC, return on carry; RM, return on minus usw.). Der Befehlszähler wird aus dem Kellerspeicher geholt, zuerst PCL, dann der Stapelzeiger um 1 inkrementiert und darauf PCH.

Schließlich wird wieder der Stapelzeiger um 1 inkrementiert. In *Abb. 120* ist dieser Vorgang für einen Rücksprungbefehl vorher und, in *Abb. 121*, danach dargestellt.

Die Sprungbefehle nach einem anderen Programmteil (nicht nach einer Subroutine, der Inhalt des Befehlszählers bleibt nicht erhalten), wird von jump-Befehlen ausgeführt. Der JMP-Befehl (jump unconditional) ist für den unbedingten Sprung vorgesehen. Auch bedingte Sprünge sind möglich, JC (jump on carry), JM (jump on minus) usw.
Sämtliche call- und jump-Befehle sind Dreibyte-Befehle. Nach dem Operationscode folgt die Sprungzieladresse in direkter Adressierung. Relative Adressierung ist hier nicht vorgesehen.

Für die bedingten Sprungbefehle nach und von einer Subroutine sind in der Spalte „Number of states" zwei Zahlen angegeben. Die kleinere Zahl gilt für den Fall, wenn der Sprung nicht, dagegen die größere, sofern der Sprung ausgeführt wird.

17 Programmiersprachen

17.1 Allgemeines

Das Programmieren in Computersprache (in binärer oder hexadezimaler Operationscode-Form) erfordert Hilfsmittel, die den Prozessor wissen lassen muß, welchen Inhalt und Adresse der betreffende Speicherplatz erhalten muß. In der einfachsten Form sind für die in Frage stehenden Prozessoren sechzehn Schalter, um eine binäre Adresse und acht Schalter, um binäre Daten zu bilden, erforderlich. Jeder Schalter hat dabei zwei Stellungen, wobei die eine dem logischen Zustand 0 und die andere dem logischen Zustand 1 entspricht. Sämtliche Schalter sind mit den Eingangstoren verbunden, so daß der Prozessor selbst dem Adressen- und Datenbus den entsprechenden Wert erteilt.

Diese Eingabeart ist wohl für Lehrzwecke geeignet, jedoch so unpraktisch und zeitraubend, daß selbst ein Anfänger schnell nach einer anderen Möglichkeit zur Programmeingabe verlangt.

Diese Möglichkeit ergibt sich durch ein hexadezimales Tastenfeld. Dieses enthält außer einer Reihe Bedienungstasten auch noch sechzehn mit 0 bis einschließlich F numerierte Tasten. Dazu gehören noch wenigstens zwei Tasten, eine zur Ankündigung, daß Daten, die andere, daß Adressen eingegeben werden. Dann ist noch ein Display mit mindestens sechs Ziffern vorhanden, davon vier für die Anzeige und zwei für den Inhalt einer Adresse.

Dadurch vereinfacht sich wesentlich die Programmeingabe, obgleich bei langen Programmen der Zeitbedarf für das Laden in den Computer erheblich ist. Die Wahrscheinlichkeit, daß Fehler auftreten, ist dabei recht groß, und zwar so groß, daß bei jedem umfangreichen Programm eine Reihe von Fehlern mit einkalkuliert werden muß. Unterstellen wir, daß die Problemlösung einwandfrei ist, dann ergeben sich Fehler u.a. beim Codieren der Mnemonic-Symbole, beim Berechnen der relativen Adressen, bei Sprungbefehlen oder beim Eintasten des Programms.

Viele Schwierigkeiten werden vermieden, wenn ein Computer mit einer Tastatur ähnlich einer Schreibmaschine, einem Videodisplay (VDU) oder einem Drucker – zum Auslesen der Speicherplätze und der Ergebnisse – und einem „Assembler" ausgerüstet ist. Ein Assembler-Programm wird dem System-Programm eines Computers beigegeben und ist in der Lage, Mnemonic-Symbole in Maschinensprache zu übersetzen. Daraus ergibt sich die Möglichkeit, ein Programm in Mnemonic-Symbolen einzugeben, der Assembler codiert die eingegebenen Symbole in Maschinensprache und das zeitraubende Codieren des Programms in Maschinencode entfällt. Der Assembler kann auch evtl. eingebrachte Fehler testen und anzeigen. Programme, die direkt in Maschinensprache unter Zuhilfenahme eines Assemblers eingegeben werden, haben jedoch folgende Nachteile:

Ein Programm kann nur für einen bestimmten Computertyp geschrieben werden. Das Programm ist „Maschinen-orientiert"; man bezeichnet die Maschinensprache deshalb auch eine „niedere Computersprache". Ein umfangreicheres Programm enthält auch eine große Anzahl von Schritten oder Zeilen. Es sind daher zahlreiche Befehle einzuführen.

Wenn von einer „niederen Computersprache" die Rede war, dann muß es auch eine „höhere Computersprache" geben. Das ist dann die „Anwender-orientierte" oder „Problem-orientierte" Programmiersprache. Computer, die für diese „Sprachen" geeignet sind, müssen auf jeden Fall mit einem umfangreichen Tastenfeld zur Eingabe der Programme ausgerüstet sein, sowie einer Auslesemöglichkeit in Form einer VDU oder einem Drucker. Oft ist diese Apparatur vom Computer getrennt aufgestellt und wird dann mit „Terminal" bezeichnet. Der Zweck dieser Programmiersprachen ist deutlich erkennbar:

Die Zahl der Programmierschritte muß so klein wie möglich gehalten werden und die Sprache sollte für alle Computer gleich sein. Außerdem sollten Fehlermeldungen ausgegeben werden können.

Leider gibt es unterschiedliche Computermodelle, so daß noch viele Unterschiede in den Programmiersprachen für die einzelnen Computer vorhanden sind. Diese sind jedoch so gering, daß die Regeln, die für eine bestimmte Computersprache gelten, deutlich erkennbar sind.

Zu den problemorientierten Sprachen gehören u.a. „FORTRAN" (*FOR*mular *TRAN*slator) für wissenschaftliche Probleme und „COBOL" (*CO*mmon *B*usiness-*O*riginal-*L*anguage) für verwaltungstechnische Aufgaben.

Eine Anwender-orientierte Sprache ist BASIC (*B*eginners *A*llpurpose *S*ymbolic *I*nstruction Code), die zu Beginn der 60er Jahre im Darthmouth College (USA) entwickelt wurde. Als vierte höhere Programmsprache ist „ECOL" (*E*ducational *C*omputer *L*anguage) erwähnenswert, zumal viele Lehrgangsteilnehmer hiermit ihre ersten Computererfahrungen sammeln.

Jede Sprache muß mit Hilfe von Hilfsprogrammen „übersetzt" werden, die den Systemprogrammen eines Computers mit der Betriebsanleitung beigegeben werden. Hier gibt es nun zwei Methoden:

a) Der „Compiler". Das eingegebene Programm wird gelesen und in einen Speicher eingelagert. Dann wird es im ganzen in ein äquivalentes Maschinenprogramm übersetzt, das dann vom Computer ausgeführt wird.

b) Der „Interpreter". Auch hier wird das eingegebene Programm gelesen und gespeichert, nur wird hier nicht das gesamte Programm auf einmal, sondern Zeile für Zeile übersetzt und ausgeführt.

Ein Compiler benötigt einen größeren Speicherumfang als ein Interpreter, letzterer arbeitet aber langsamer, besonders, wenn das Programm eine Reihe von Subroutinen enthält. Jedes Mal, wenn eine Subroutine aufgerufen wird, muß diese wieder übersetzt werden. Infolge seiner allgemeinen Verwendbarkeit und da die Sprache recht einfach zu lernen ist, hat BASIC eine große Verbreitung als Programmiersprache für Mikrocomputer erfahren. Da ein Mikrocomputer nur einen beschränkten Speicherumfang besitzt und der Speicher derzeit noch den höchsten Kostenfaktor für einen Mikrocomputer darstellt, wird oft eine vereinfachte BASIC-Ausführung, die TINY BASIC, eingesetzt.

In den meisten Fällen wird vom Computer-Hersteller ein Interpreter geliefert. Dieser ist natürlich nur für die bestimmte Computertype geeignet (in Verbindung mit den Unterschieden in der Maschinensprache) und entspricht den Möglichkeiten, die der Hersteller in die Computertype eingebaut hat. Daß der Interpreter ziemlich langsam arbeitet, braucht zu keinen Bedenken Veranlassung zu geben, solange der Computer auch die Möglichkeit bietet, in Maschinensprache zu programmieren. Programme, die sowieso schnell laufen

sollen, können dann in Maschinensprache geschrieben werden, indem man von den Möglichkeiten, die der Prozessor bietet, Gebrauch macht. Gerade das Programmieren in Maschinensprache ist für den Programmierer die interessanteste Programmiermethode.

17.2 Die Programmiersprache BASIC

Die hier folgende „Beschreibung" von BASIC ist nur zum Kennenlernen gedacht, denn das Programmierenlernen in BASIC ist nur mit Hilfe eines hierfür geeigneten Computers möglich.

Beim Arbeiten mit einem Computer gibt es so etwas wie einen persönlichen Kontakt zwischen Computer und Anwender. Der Computer stellt Fragen, gibt Aufträge an den Anwender und dieser gibt darauf Antwort. In Wirklichkeit ist das natürlich nicht wahr, dazu ist die Maschine viel zu dumm; Fragen und Aufträge sind ja im Programm eingearbeitet, das bearbeitet werden soll. Ganz allgemein kann unterstellt werden, daß der Computer weniger intelligent ist als das ihm eingegebene Programm.

Vorläufig gehen wir davon aus, daß zwischen Computer und Anwender eine gewisse Verbindung besteht. Diese läuft über eine Tastatur, eine VDU oder einen Drucker. Daher kennt BASIC auch eine Anzahl von „Kommandos", die sich auf diese Kommunikation beziehen und außer diesen auch noch Kommandos, die sich auf das Programmieren beziehen. Diese „Kommandos" werden in englischen Worten bzw. Abkürzungen erteilt. Die Kommandos, die sich auf das Programmieren beziehen, werden mit „Statements" bezeichnet.

Wie das Übersetzungsprogramm bei einem Computer nach dem Einschalten gestartet wird, ist von Computer zu Computer verschieden und geht aus der mitgelieferten Betriebsanleitung hervor. Darin ist auch die Ankündigung des Computers enthalten, ob er für die Information empfangsbereit ist. Häufig ist das letzte Wort dieser Meldung „OK" oder „READY", je nach Computertype. Am folgenden Zeilenbeginn steht z.B. eines der aufgeführten Zeichen >, □ oder #. Diese werden „Prompt" genannt und bedeuten, daß der Computer für eine Informationsaufnahme bereit ist. Sämtliche Lese-, Satz- und Interpunktionszeichen, die vom Anwender auf der Tastatur angeschlagen werden, stehen dann, wie bei einer Schreibmaschine, auf einer VDU oder werden mit einem Drucker wiedergegeben, und zwar auf der Zeile, die mit dem Prompt gekennzeichnet ist. Weiterhin finden wir auf der VDU einen „Cursor" („–" oder dergleichen), der angibt, wo das nächste Zeichen auf dem Schirm steht.

Als Einleitung läuft folgendes kleines Programm ab:

```
NEW
10 LET A = 3 + 5
20 PRINT A
30 END
RUN
```

Das erste eingetastete Wort ist „NEW". Damit wird vermieden, daß ein gerade vorher eingegebenes Programm den Ablauf des neuen Programms stört. Alte Programme werden damit gelöscht. NEW ist daher ein „Kommando". Nach NEW muß die Taste RETURN

oder ENTER gedrückt werden. Damit wird das nächste Zeichen an den Anfang der nächsten Zeile geschrieben und an das Übersetzerprogramm (Interpreter) das Ende der Kommando- oder Programmzeile gemeldet. Nun folgt die erste Programmzeile:

```
10 LET A = 3 + 5
```

Jede Programmzeile erhält eine Zeilennummer. Das Programm wird nämlich nicht nach der Reihenfolge, wie es eingetastet wird, sondern in der Reihenfolge der Zeilennummer, abgearbeitet. Vorzugsweise beginnen die Zeilennummern mit dem Faktor 10, es ergibt sich dann später die Möglichkeit, zwischen zwei Zeilen eine weitere einzufügen. Diese neue Zeile kann dann z.B. am Programmende eingetastet werden. Der Übersetzer bringt dann diese Zeile in Abhängigkeit von der Zeilennummer an seinen Ort.

In Zeile 10 ist A eine „Variable", das bedeutet, daß A einen beliebigen Inhalt erhalten kann. Sobald in einem Programm mehr als 26 Variable „umgetauft" werden müssen, kann die Kennzeichnung oder der „Name" aus zwei Buchstaben bestehen. In dieser Zeile bedeutet LET A = 3 + 5, daß der Inhalt der Variablen A mit der Summe von 3 und 5 übereinstimmen muß.
Die nächste Programmzeile ist:

```
20 PRINT A
```

Das Ergebnis der Berechnung soll auf einem Schirm oder Drucker abgebildet werden. Dazu dient das Statement PRINT A mit dem Auftrag, den „Wert" A zu drucken. Das letzte Statement

```
30 END
```

bedeutet einfach Programmende.

Das gesamte Programm besteht aus drei Zeilen, die nunmehr eingetastet sind. Jetzt ist noch ein Kommando erforderlich, um den Computer zum Abarbeiten des Programms zu veranlassen, und zwar „RUN". Damit wird das Ergebnis gedruckt.
Das Programm wird, wie folgt, eingegeben (R = RETURN)

```
NEW R
10 LET A = 3 + 5 R
20 PRINT A R
30 END R
RUN R
```

Gleich nach dem letzten RETURN wird die Antwort gedruckt. Dieses Programm kann auch als Unterprogramm angesehen werden, so daß der Wert A (3 + 5 = 8) weiter im Programm verarbeitet werden kann. Soll das Ergebnis jedoch lediglich sichtbar gemacht werden, dann heißt das Programm:

```
NEW
10 PRINT 3 + 5
20 END
RUN
```

Es kann aber auch nach dem Statement PRINT eine weitere Rechnung folgen. Es fehlt aber ein Speicherplatz mit Inhalt 8, so daß das Ergebnis für weitere Berechnungen nicht zur Verfügung steht.

Im nächsten Programm werden die Kommandos NEW und RUN nicht mehr angegeben, da sie sowieso eingesetzt werden müssen. Das gleiche gilt für die Taste RETURN, die selbstredend nach jeder Zeile zu betätigen ist.

Wie es in Maschinensprache verschiedene Adressierungsarten gibt, so ist es auch möglich, auf verschiedene Arten Daten (hier 3 und 5) einzugeben, wie es im nächsten Beispiel das READ-Statement angibt:

```
10 READ X, Y
20 DATA 3, 5
30 LET A=X + Y
40 PRINT A
50 END
```

Hierin sind X und Y Variable, die mit Daten aufzufüllen sind, X mit 3 und Y mit 5. In Zeile 30 findet dann die Addition 3 + 5 statt.

Nach der dritten Methode mit Statement INPUT werden die Daten nicht in das Programm eingegeben:

```
10 INPUT X, Y
20 LET A=X + Y
30 PRINT A
40 END
```

Auch hier müssen die Variablen X und Y mit Daten aufgefüllt werden. Sie werden aber nicht im Programm aufgeführt, sondern auf Kommando RUN fragt der Computer (mit Fragezeichen „?“) nach den Daten. Nun können die Daten eingetastet werden; ist das geschehen, gibt nach RETURN der Computer die Antwort: „8“. Das ist eine Art, um den Computer auf „anständige“ Weise nach Daten anfragen zu lassen. Zeile 10 des letzten Programms muß dann, wie folgt, geändert werden:

```
10 INPUT 'WAS SIND X UND Y'; X, Y
```

Nach dem RUN-Kommando wird gedruckt:

```
WAS SIND X und Y ?
```

Das Fragezeichen wird vom Computer sinngemäß zugefügt, und nach Eingabe von 3, 5 erscheint die Antwort „8“. Wesentlich sind die Anführungszeichen in Zeile 10, die auf keinen Fall vergessen werden dürfen.

Auch die Antwort kann etwas „freundlicher“ wiedergegeben werden, etwa so:

```
A = 8
```

Zeile 30 wird dafür in

30 PRINT 'A =', A

Alles, was zwischen Anführungszeichen hinter PRINT eingegeben wird, wird buchstabengetreu gedruckt. Alles, was ohne Anführungszeichen hinter PRINT eingegeben wird, wird als Variable angesehen, dessen Inhalt abgedruckt werden muß. Die Kommas dienen dabei als Trennungszeichen. Jetzt ist das Ergebnis:

A = 8

Der große Abstand zwischen 'A =' und '8' kann vermieden werden, wenn als Trennungszeichen nicht ',' , sondern ';' genommen wird:

30 PRINT 'A =' ; A

Daraus resultiert:

A = 8

Mit BASIC können folgende Rechenaufgaben ausgeführt werden:

Aufgabe	*Mathemat. Symbol*	*BASIC-Symbol*
Addition	+	+
Subtraktion	–	–
Multiplikation	x	*
Division	:	/
Potenzieren	x^y	X ↑ Y
Wurzelziehen	$\sqrt[y]{x}$	X ↑ (1/Y)

Wurzelziehen wird auf Potenzieren zurückgeführt:

$$\sqrt[y]{x} = x^{\frac{1}{y}}$$

Die Quadratwurzel einer Zahl wird mit dem Statement

SQR(X)

gezogen. Kommen mehrere Aufgaben ohne Klammersetzung in einer Berechnung vor, dann wird bei BASIC folgende Rangordnung eingehalten:

1. Potenzieren
2. Multiplikation und Division in der Reihenfolge, wie sie in der Rechnung vorkommen:

16*4/2=64/2=32
16/4*2=4*2=8

3. Addition und Subtraktion in der Reihenfolge gemäß der Rechnung:

16+4−2=20−2=18
16−4+2=12+2=14

Aufgaben zwischen den Klammern gehen vor:

16−(4+2)=16−6=10
16/ (4*2)=16/8=2

Im nächsten Programm ist zu berechnen: $R=2\sqrt{3^2+4^2}=10$.

```
10PRINT 'R='; 2*SQR(3↑2+4↑2)
20 END
```

Das Ergebnis ist:

R = 10

Außer den Hauptrechenaufgaben bearbeitet BASIC auch noch eine Reihe von Funktionen:

BASIC Symbol	*Mathemat. Symbol*
SIN(X)	sin x
COS(X)	cos x
TAN(X)	tg x
ATAN(X)	arctg x
LOG(X)	ln x
LGT(X)	log x
EXP(X)	e^x
SQR(X)	$\sqrt{x}$
ABS(X)	$\lvert x \rvert$
INT(X)	integer x

SIN(X), COS(X) und TAN(X) sind normale Winkelfunktionen. X muß hier im Bogenmaß angegeben werden. In gleicher Weise wird bei ATAN(X) der Winkel in Bogenmaß gefunden, wenn der Tangens des Winkels gegeben ist. Bei LOG(X) ist die Basis e (2,7182818) und bei LGT(X) 10. Die nächsten drei Statements sprechen für sich selbst. Mit Abrunden von x ist die Zahl vor dem Komma gemeint.

x = 23,56 INT(X) = 23

Aber:

x = −23,56 INT(X) = −24

Mit INT (X) wird „nach unten" auf eine ganze Zahl abgerundet.
Unbedingte Sprungbefehle können mit dem Statement GOTO gegeben werden.

```
10 INPUT 'WAS BEDEUTET X UND Y'; X, Y
10 INPUT 'WAS SIND X UND Y'; X, Y
20 PRINT 'A='; X+Y
30 GOTO 10
40 END
```

Nach dem Kommando RUN fragt der Computer:

```
WAS SIND X UND Y ?
```

und nach dem Eintasten von 3, 5 folgt:

```
A = 8
WAS SIND X UND Y ?
```

Wieder können zwei Werte für X und Y eingetastet werden, worauf das Ergebnis und wieder die Frage nach den Daten folgt. Das geht so lange, immer wird nach der Zeile 10 zurückgesprungen, wodurch die Zeile 40 überflüssig wird. Im nächsten Programm wird das READ-Statement eingesetzt.

```
10 READ X, Y
20 DATA 3,5,7,4
30 PRINT 'A='; X+Y
40 GOTO 10
50 END
```

Hier ist die Zeile 50 eigentlich überflüssig. Zuerst werden 3 und 5 auf X bzw. Y eingelesen und in der zweiten Runde 7 und 4. Als Ergebnis erscheint:

```
A = 8
A = 11
```

Danach folgt eine Fehlermeldung, da keine Daten mehr zur Verfügung stehen. Für einen bedingten Sprung ist das IF-THEN-Statement erforderlich:

```
10 LET X=0
20 PRINT 'X='; X
30 LET X=X+1
40 IF X < 6 THEN 20
50 END
```

Für dieses Programm wird ein Zähler benötigt. Immer wenn die Schleife durchlaufen wird, muß in Zeile 30 der Zähler 'X' um 1 inkrementiert werden. Zeile 40 verursacht einen Sprung nach Zeile 20, so lange der Wert von X kleiner als 6 ist. Bei jedem Schleifendurchgang wird der Zählerinhalt gedruckt. Das Ergebnis ist dann:

```
X = 0
X = 1
X = 2
X = 3
X = 4
X = 5
```

Erfüllt X nicht die Bedingung, dann wird auf die nächste Programmzeile übergegangen. Nachdem das Ergebnis X = 5 gedruckt ist, wird in Zeile 30 X um 1 inkrementiert auf 6, erfüllt aber nicht mehr die Bedingung x < 6. Dann folgt Zeile 50 als Anzeige des Programmendes.
BASIC kennt verschiedene Sprungbedingungen:

Bedingung	*Symbol*
gleich	=
ungleich	<>
größer als	>
kleiner als	<
größer als oder gleich	>=
kleiner als oder gleich	<=

Hierbei kann der Wert einer Variablen mit einer Zahl, mit einer anderen Variablen oder mit einem arithmetischen Ausdruck verglichen werden.

```
40 IF  X< 6 THEN 20
40 IF  X=Y  THEN 20
40 IF  X> Y*(Y-1)↑2 THEN 20
40 IF  X↑(1/3)*(X-1)< 10 THEN 20
```

Ein Zähler, wie im vorigen Programm, kommt oft vor. Es gibt dafür ein Statement, das selbst einen Zähler beinhaltet, so daß dieser nicht besonders programmiert zu werden braucht:

```
10 FOR  X=0  TO 5
20 PRINT 'X='; X,2*(X+1)
30 NEXT  X
40 END
```

In Zeile 10 werden Anfang und Ende des Zählers definiert (0 und 5). Bei jedem Schleifendurchgang wird der Zähler um 1 erhöht. In Zeile 20 wird zuerst X = gedruckt, dann der X-Wert und endlich das Ergebnis der Berechnung 2 * (X+1). Zeile 30 erteilt den Sprungbefehl nach Zeile 10, in der der Zähler wieder um 1 inkrementiert wird. Da hier nur eine Variable vorhanden ist, sollte

```
30 NEXT
```

genügen. Bei mehr als einer Variablen sollen jedoch immer die betreffenden Variablen genannt werden.

Wenn X = 5 ist, findet kein Sprung statt. Dann wird Zeile 40 abgearbeitet. Das Ergebnis aus diesem Programm ist:

```
1. Durchgang:  X=0      2
2. Durchgang:  X=1      4
3. Durchgang:  X=2      6
4. Durchgang:  X=3      8
5. Durchgang:  X=4     10
6. Durchgang:  X=5     12
```

Es sind auch größere Schritte möglich:

```
10 FOR X = 0 TO  10 STEP  2
```

STEP 2 bedeutet für den Zähler, daß dieser um 2 inkrementiert werden muß. Anfang und Ende des Zählers können willkürlich definiert werden.

```
10 FOR  X=10  TO    25 STEP     5
10 FOR  X=10  TO   -25 STEP    -5
```

Im zweiten Beispiel wird der Zähler immer um 5 erniedrigt. Im-nächsten Programm kommen drei Variable zur Anwendung:

```
10  READ A,B,C
20  DATA 3,5,7
30  PRINT "R="; A+B+C
40  END
```

Für jede Variable ist ein Speicherplatz zu reservieren und jedem ein entsprechender „Name" zugeteilt (A, B oder C). Es ist aber auch möglich, eine Reihe von Variablen mit demselben Buchstaben zu kennzeichnen, die durch ein „Subscript" (einen Index) deutlich zu machen sind, z.B.

A(1), A(2), A(3) usw.

allgemein:

A(X) (X ist das „Subscript").

In unserem Beispiel sind drei Variable erforderlich: A(0), A(1) und A(2). Diese Zusammenstellung wird ein „Array" genannt. Es ist dies ein eindimensionales Array. Bei einem zweidimensionalen Array werden die Elemente, wie folgt, gekennzeichnet:

A(2,3), A(1,5), A(3,7) usw.

allgemein:

A (X,Y).

Zuvor muß jedoch bekannt sein, wieviele Speicherplätze für den Array vorzusehen sind. Hierzu dient das Statement „DIM" (von „Dimension"). In unserem Beispiel müssen drei Speicherplätze vorgesehen werden. Das Programm ergibt sich, wie folgt:

```
 5 DIM A(3)
10 READ A(0), A(1), A(2)
20 DATA 3,5,7
30 PRINT "R="; A(0)+A(1)+A(2)
40 END
```

Im Prinzip ist das Programm gleich geblieben. Lediglich die Bezeichnung der Variablen wurde geändert und Zeile 5 zugefügt.

Das „Subscript" kann ebenfalls eine Variable sein:

```
10 DIM A(5)
20 FOR X=0 TO 4
30 READ A(X)
40 DATA 3,4,5,7,9
50 PRINT "A="; A(X), A(X)*[A(X)+1]
60 NEXT X
70 END
```

Hier werden fünf Speicherplätze für die in Zeile 40 angegebenen Zahlen reserviert. Auf Zeile 30 wird hintereinander 3, 4, 5, 7 und 9 eingelesen. Der Index erhält hier den Wert der Variablen X, die bei jeder Schleife erhöht wird. Daraus ergibt sich folgendes Ergebnis:

```
A=3          12
A=4          20
A=5          30
A=7          56
A=9          90
```

Auch in BASIC kann ein Unterprogramm aufgerufen werden, und zwar mit Hilfe des Statements GOSUB. Nach diesem Statement muß die Zeilen-Nummer des ersten Statements der Subroutine folgen. Am Ende der Subroutine muß das Statement RETURN eingesetzt werden. Dadurch wird nach dem Durchlauf des Unterprogramms mit dem Hauptprogramm wieder auf die Stelle gegangen, wo sie verlassen wurde. Ein Beispiel: Es soll das Gewicht eines dickwandigen, würfelförmigen Behälters mit einer Außenabmessung von 25 cm (Kante) und einem Innenmaß von 20 cm (*Abb. 122*) berechnet wer-

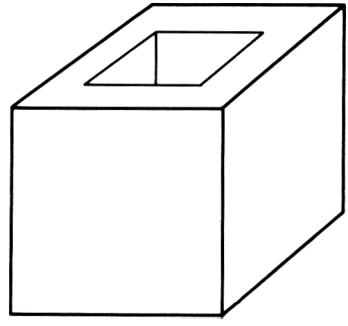


Abb. 122 Körper, dessen Gewicht berechnet werden soll

den. Das Material des Behälters hat ein spez. Gewicht von 7 g/cm^3. Hierfür eignet sich das nächste Programm in BASIC:

```
10 LET X=25
20 GOSUB 80
30 LET A=I
40 LET X=20
50 GOSUB 80
60 PRINT A–I
70 END
80 LET I=X↑3*7
90 RETURN
```

Da zweimal ein Gewicht aus I=X ↑ 3*7 berechnet werden muß, gebrauchen wir hierfür die Subroutine auf den Zeilen 80 und 90. Erst wird das Gewicht des massiven Würfels berechnet (A) und danach das imaginäre Gewicht des Innenraums (I). Der Unterschied zwischen A und I ergibt das geforderte Gewicht.

Genau besehen, hat es hier keinen Sinn, eine Subroutine einzusetzen. Besser geeignet ist ein FN- und ein DEF-Statement. Das gibt uns die Möglichkeit, eine Funktion aufzustellen und dort einzusetzen, wo dies nötig ist.

```
10 DEF FNG(X)=X↑3*7
20 LET A=FNG(25)
30 PRINT A–FNG(20)
40 END
```

Zeile 10: Hier wird die Funktion definiert. Das Statement dafür ist DEF FN; mit einem Buchstaben der Funktion kann eine Bezeichnung gegeben werden (hier G). Es sind max. 26 Funktionen zu bezeichnen.

Zeile 20: Hier wird 25 ↑ 3*7 errechnet, indem X den Wert 25 erhält.

Zeile 30: Hier wird der Unterschied vom Gewicht des massiven Würfels (A) und dem imaginären Gewicht des Innenraums (20↑3*7) berechnet.

Im nächsten Programm wird das Endresultat einer Verzinsung ermittelt, wenn pro Jahr 6,25% Zinsen auf das angelegte Kapital aufgeschlagen werden.

```
10 DEF FNI(X)=X+X*6.25/100
20 INPUT 'WIE HOCH IST DER BETRAG'; A
30 INPUT 'WIEVIELE JAHRE'; B
40 FOR P=1 TO B
50 LET A=FNI(A)
60 PRINT 'JAHR'; P,A
70 NEXT P
80 END
```

Zeile 10: Definition der Funktion. Die Zinsen des Jahresbetrages werden zum Jahresbetrag addiert.

Zeile 20: Nach RUN fragt der Computer an, wie groß das angelegte Kapital ist.

Zeile 30: Der Computer fragt nach der Anzahl der Jahre.

Zeile 40: Der Anfangsstand des Zählers ist 1 und der Endstand gleich der Anzahl der Jahre.

Zeile 50: Hier wird pro Jahr der Endbetrag berechnet.

Zeile 60: Der Computer druckt die Jahreszahl und den Endbetrag von diesem Jahr aus.

Zeile 70: Das Ergebnis ist eine Reihe von Zahlen, die den Endbetrag pro Jahr angeben.

Ist die Höhe des angelegten Kapitals DM 2500,– und die Anzahl der Jahre 6, dann soll von einem Drucker oder einer VDU das Ganze wie folgt dargestellt werden:

```
NEW
10 DEF FNI(X)= X+X * 6.25/100
20 INPUT "WIE HOCH IST DER BETRAG"; A
30 INPUT "WIEVIELE JAHRE"; B
40 FOR P = 1 TO B
50 LET A = FNI(A)
60 PRINT "JAHR"; P,A
70 NEXT P
80 END
RUN
WIE HOCH IST DER BETRAG?
2500
WIEVIELE JAHRE ?
6
JAHR 1          2656.25
JAHR 2          2822.27
JAHR 3          2998.66
JAHR 4          3186.07
JAHR 5          3385.2P
JAHR 6          3596.78
```

Nach dem Kommando RUN (gefolgt durch RETURN) arbeitet der Computer das Programm bis einschließlich Zeile 20 ab. Nun muß der Betrag 2500 eingetastet werden (auch wieder mit nachfolgendem RETURN). Der Computer setzt das Programm mit Zeile 30 fort. Nach Eintasten der Zahl 6 (RETURN) arbeitet der Computer das weitere Programm ab. Für jede Variable wird ein Speicherplatz bereitgestellt. Wird die nächste Programmzeile eingegeben

```
10 LET  A=5
```

dann wird ein Speicherplatz reserviert mit der Bezeichnung A und Inhalt 5. Jedesmal, wenn der Computer im Programm später etwas mit der Variablen A zu tun hat, wird immer der Inhalt dieses Speicherplatzes gelesen. Die Anzahl Ziffern, die pro Variable eingegeben werden kann, ist von Computer zu Computer verschieden.

Eine andere Art von Variablen ist die, bei der der Inhalt nicht aus einer Anzahl Ziffern, sondern aus alphanumerischen Zeichen besteht. Diese Variable wird mit einem Buchstaben aus dem Alphabet, mit dem Kennzeichen „$“ und als „String“ bezeichnet.

Das nächste kleine Programm

```
10 LET A$ = „BASIC-PROGRAMM "
20 PRINT A$; „ERSTER TEIL"
```

hat als Ergebnis:

BASIC-PROGRAMM ERSTER TEIL

Es wird auf den Zwischenraum zwischen „PROGRAMM" und „ERSTER", einprogrammiert in Zeile 10, aufmerksam gemacht. Dieser ist notwendig, da beim Statement PRINT in Texten vor und nach dem Trennungszeichen kein Zwischenraum vom Computer ausgegeben wird.

Befehl: 10 PRINT "A="; "5"
Ergebnis: A= 5
Befehl: 10 PRINT "FOTO"; "GRAF"
Ergebnis: FOTOGRAF

Beim Drucken von Zahlen spielen die Zeichen „ ," und „ ;" eine wesentliche Rolle. Das nächste Programm:

```
10 LET X=0
20 PRINT X
30 LET X=X+1
40 IF X < 5 THEN 20
50 END
```

hat das Ergebnis:

```
0
1
2
3
4
```

Jetzt wird Zeile 20 mit einem „ ," abgeschlossen:

```
10 LET X=0
20 PRINT X,
30 LET X=X+1
40 IF X < 5 THEN 20
50 END
```

Das führt dazu, daß die Zahlen in einer Zeile gedruckt werden, wobei die Länge der Zeile in „Zonen" aufgeteilt ist.

```
0   1   2   3   4
```

Zeile 20 kann auch mit einem „ ;“ abgeschlossen werden:

```
10 LET X=0
20 PRINT X;
30 LET X=X+1
40 IF X < 5 THEN 20
50 END
```

Auch jetzt werden die Zahlen in eine Zeile gedruckt, jedoch enger beieinander.

```
0 1 2 3 4
```

Die Anwendung des Kommas bei Zahlen teilt die Zeile in Zonen ein, deren Größe von der Zahl der Ziffern abhängt, aus der die Zahl besteht.

Anzahl Ziffern der Zahl	*Anzahl der Abstände in der Zone*
1	3
2, 3, oder 4	6
5, 6 oder 7	9
8, 9 oder 10	12
11	15

Es kann sich als notwendig erweisen, in ein Programm Erklärungen aufzunehmen. Es muß dann ein Text eingeschrieben werden, der keinen Vorgang im Programm hervorruft. Hierzu dient das Statement REM, abgeleitet von REMARK.

```
10 REM  BERECHNUNG KOORDINATEN
20 LET  A=0.7
30 LET  B=15
40 LET  X=B*COS(A)
50 LET  Y=B*SIN(A)
60 PRINT  X, Y
70 END
```

Zeile 10 hat keinen Einfluß auf das Programm und wird auch nicht im Ergebnis abgedruckt.

Wird der Computer als „Spielcomputer“ eingesetzt, dann gibt es hierfür ein RND-Statement (RND von RANDOM). Mit diesem Statement erzeugt der Computer eine RANDOM-Zahl (beliebige Zahl) zwischen 0 und 1 mit einer Anzahl Dezimalen, abhängig vom eingesetzten Computer. Auch die Art, wie man auf eine größere Zahl gelangen kann, ist vom Computer abhängig. Bisweilen ergibt der Befehl:

```
10 LET  A = RND(X)
```

eine Zahl zwischen Null und X (X = 20, A = zufällige Zahl zwischen Null und zwanzig). Bei wieder anderen Computern kann für X eine beliebige Zahl gewählt werden. Das Er-

gebnis ist dann, abhängig vom Wert von X, eine beliebige Zahl zwischen 0 und 1. Der Befehl:

```
10 LET A = RND(X)*20
```

erzeugt dann eine Zufallszahl zwischen Null und zwanzig. Die Zahl zwanzig ist hierbei nicht inbegriffen. Es ist auch möglich, nur ganze Zahlen zwischen Null und zwanzig zu erzeugen:

```
10 LET A = INT [RND(X)*20]
```

Die Zahl Null ist hier inbegriffen. Ist dies jedoch nicht erforderlich, dann wird „0" wie folgt ausgeklammert:

```
10 LET A = INT [RND(X)*20]
20 IF A = 0 THEN 10
30 nächstes Statement
```

Wenn A „0" ist, wird wieder eine Zufallszahl erzeugt. Dies setzt sich so lange fort, bis für A eine Zahl, die nicht Null ist, gefunden wird.

Auch bei BASIC kann mit Hilfe einer Schleife eine Verzögerung hervorgerufen werden:

```
10 LET X = 0
20 LET X = X + 1
30 IF X < A THEN 20
40 nächstes Statement
```

Die Verzögerungszeit ist der Größe der Zahl A proportional. Auch eine FOR-NEXT-Schleife wäre dafür geeignet.

Die wichtigsten Statements von BASIC sind hiermit wohl beschrieben; nur eines sollte noch erwähnt werden, und zwar, um den Computer auf einer bestimmten Stelle im Programm anhalten zu lassen, das Statement STOP. Bringen wir dieses z.B. auf Zeile 35 eines Programms, dann wird nach RUN dieses Programm bis zur Zeile 35 durchlaufen. Der Computer hält an und meldet z.B.:

```
BREAK IN LINE 35
```

Mit dem Kommando GO wird das Programm fortgesetzt und mit RUN beginnt der Computer wieder von vorn.

Von den Kommandos ist noch das LIST-Kommando zu erwähnen. Dieses druckt das Programm, so wie es sich im Speicher befindet, aus. Wurde es z.B. in folgender Reihenfolge eingetastet:

```
10 Statement a
20 Statement b
15 Statement c
30 Statement d
```

dann erfolgt die Wiedergabe nach LIST:

10 Statement a
15 Statement c
20 Statement b
30 Statement d

Es ist auch möglich, Teilprogramme zu drucken:

LIST 70–150

Das Programmteil zwischen den Zeilen 70 und 150 wird nunmehr gedruckt.
Entsprechend dem Computer-Fabrikat gibt es noch weitere Kommandos, die von BASIC verarbeitet werden können. Auf diese Kommandos wird in der betreffenden Computer-Betriebsanleitung hingewiesen.

18 Wortregister

Akkumulator	CPU-Register für arithmetische und logische Operationen.
A/D-Konverter	Analog/Digital-Wandler.
Adreßbus	Anzahl Parallel-Leitungen für Adreßcode-Transport.
Algorithmus	Siehe „Programm".
Arithmetical instruction	Befehl zum Ausführen von arithmetischen Operationen.
Assembler	Software, um mnemonischen Code in Maschinen-orientierte Programmsprache umzusetzen.
Assembly Language	Sammlung symbolischer Ausdrücke für Programmbefehle.
Asynchron	Taktunabhängige Operationen zum Starten einer neuen Operation, sobald diese beendet sind.
Baudot-Code	Fünfelement-Code für alphanumerische Daten.
Bidirektional	Arbeitet in zwei Richtungen, z.B. der Datenbus.
Binärsystem	Zweiwertiges System.
Branch	Befehl für einen Programmsprung (Verzweigung).
Buffer	Schaltung, die Datensignale wieder auf Sollpegel regeneriert.
Bug	Fehler im Programm.
Byte	Binärwort, bestehend aus acht Bits.
Carryflag	Bit des Statusregisters; wird 1, wenn der Übertrag einer arithmetischen Operation 1 wird.
Character	Symbol für Buchstaben, Ziffern, Satzzeichen usw. (Zeichen).
Clock generator	Elektronische Schaltung für die Taktimpuls-Erzeugung.
Compiler	Übersetzungsprogramm, um eine höhere Programmiersprache in Maschinensprache umzusetzen.
D/A-Konverter	Digital/Analog-Umsetzer.
Data	Daten, Information (engl.)
Datenbus	Anzahl paralleler Verbindungsleitungen zum Datentransport.
Debugging	Aufsuchen und Korrigieren von Programmfehlern.
Destination register	Zielregister.
Entwicklungssystem	Computer, mit dem ein Programm entwickelt werden kann.
Execute	Ausführen einer Anzahl von Programmschritten.
Flag	Statusregisterbit oder „Merkzelle".
Floppy Disk	Hintergrundspeicher; Platte mit Magnetschicht, in die Daten eingeschrieben werden können.
Flow chart	Flußdiagramm.
Hardware	Mechanische, magnetische, elektronische und elektrische Bausteine eines Computer-Systems.
Hexadezimal-System	Zahlensystem zur Basis 16 mit den Ziffern 0 . . . 9, A, B, C, D, E, F.
Hintergrundspeicher	Speicher außerhalb des Computers für ständiges Speichern eines Programms.

Input/output device	Baustein zum Datenaustausch zwischen Computer und Peripheriegerät.
Instruction	Operation, ggf. mit Operand oder Operandenadresse.
Instruction set	Befehlssatz, der vom Computer ausgeführt werden kann.
Interface	Schaltung, die verschiedene Bausteine eines Systems miteinander verbindet und einander anpaßt.
Interpreter	Hilfsprogramm zum Ausführen der Befehle einer höheren Programmiersprache.
Interrupt	Programmunterbrechung, von der aus nach Ausführen eines Interrupt-Programms das ursprüngliche Programm vom Unterbrechungspunkt ab, weiter abgearbeitet werden kann.
Kansas-City-Format	Definition eines Kassetten-Interface-Systems mit 300 Baud.
Latch	Einfaches Speicherelement (Auffang-Flipflop).
Maschinencode	Computer-Befehle in Binär-Code.
Mask	Bitmuster, das zusammen mit einer logischen Operation ein oder mehrere Bits aus einem anderen Binärwort ausblendet.
Mikroprozessor	Zentrale Prozessoreinheit, auf einem Chip integriert.
Mnemonic-Symbol	Symbol für eine Operation, um das menschliche Gedächtnis zu unterstützen (z.B. LDA für „Load Accu").
Modem	Modulator-Demodulator.
Nibble	4-Bit-Datenwort, halbes Byte.
Non-volatile	Nicht flüchtig; Daten bleiben erhalten, auch nach Spannungsabschaltung.
Oktal-System	Zahlendarstellung mit der Basis 8; Ziffern 0 . . . 7.
Operand	Argument, das vom Computer verarbeitet wird.
Operation	Operation, die vom Computer nach Befehl ausgeführt wird.
Operationscode	Befehlsteil in binärer Form zur Darstellung einer Operation.
Parallel mode	Datentransport über mehrere Leitungen, bei der Bit gleichzeitig eingegeben werden.
Parity	Methode, um die richtige Übertragung binärer Zahlen zu kontrollieren (Gleichheit).
Personal Computer	Computer für den Privatgebrauch.
Programm	Gesamtheit aufeinanderfolgender Operationen, die auf ein gewünschtes Ziel hinführen.
Prompt	Zeichen gibt an, daß Computer für den Empfang des nächsten Befehls bereit ist.
Periphere Einheit	Gerät zum Ausgeben oder Empfangen von Daten nach oder vom Computer.
Random generator	Zufallsgenerator.
Read	Daten auf Speicher oder Eingangstor lesen.
Serial	Datentransport, bei dem Bit nacheinander über eine Leitung gegeben werden.
Singleboard-Computer	Prozessor, Speicher, Ein/Ausgang, ggf. Tastatur und Display auf einer Platine untergebracht.
Single byte instruction	Programmbefehl, bestehend aus nur einem 8-Bit-Code.
Single chip computer	Prozessor, Speicher und Ein/Ausgang auf einem Chip integriert.
Stack	Last-in, first-out Speicher.

Statement	Programmbefehl.
Terminal	1. Gerät zur Kommunikation zwischen Anwender und Computer. Besteht aus Eingabegerät (z.B. Tastatur) und Ausleseeinheit (VDU oder Drucker). 2. Anweisung für Beginn und Ende eines Flußdiagramms.
Tetrade	Die niedrigsten oder höchsten vier Bit eines Byte.
Three-byte-instruction	Programmbefehl, bestehend aus drei Acht-bit-codeworten.
Tristate-Buffer	Puffer mit drei Ausgangszuständen: „0", „1" und hochohmig.
Zweierkomplementsystem	Rechenmethode für binäre negative Zahlen.
Two-byte instruction	Programm-Befehl, bestehend aus zwei Achtbitcodeworten.
Vektor	Speicherzellen, die eine Adresse enthalten.
Volatile	Flüchtig. Daten gehen verloren bei Spannungsabschaltung.
Wort	Binäre Zahl (geschlossene Bitgruppe).
Write	Daten in Speicherplatz oder Ein/Ausgangstor einschreiben.
Zero flag	Statusregisterbit; wird 1, wenn das Ergebnis einer Operation Null ist.

19 Abkürzungen

A-D	Analog to Digital
ALGOL	Algorithmic Language
ALU	Arithmetic and Logic Unit
ANSI	American National Standard Institute
APL	A Programming Language
ASCII	American Standard Code for Information Interchange
ASR	Automatic Send-Receive
BCD	Binary Coded Decimal
BCS	British Computer Society
BETA	Business Equipment Trade Association
BIT	Binary digit
CAD	Computer Aided Design
CAT	Computer Average of Transients
CCIT	Consultative Committee Int. Telephone & Telegraph
COBOL	Common Business Oriented Language
COM	Computer Output to Microfilm
CMOS	Complementary Metal Oxide Silicon
CPU	Central Processor Unit.
CR	Carriage Return
CUTS	Computer Users Tape System
D-A	Digital to Analog
DCE	Data Control Equipment
DDC	Direct Digital Control
DIL	Dual in Line

DMA	Direct Memory Access
DOS	Disc Operating System
DTL	Diode Transistor Logic
EAROM	Electrically Alterable Read Only Memory
ECMA	European Computer Manufactures Association
ECOL	Educational Computer Language
EDP	Electronic Data Processing
EPROM	Erasable Programmable Read Only Memory
ESONE	European Standard of Nuclear Electronics
FORTRAN	Formular Translator
FSK	Frequency Shift Keying
I-O	Input-Output
IRIG	Inter Range Instrumentation Group
KSR	Keybord Send-Receive
LED	Light Emitting Diode
LF	Line Feed
LSB	Least Significant Bit
LSI	Large Scale Integration
MICR	Magnetic Ink Character Recognition
MOS	Metal Oxide Silicon
MPS	Multi Processor system
MSB	Most Significant Bit
MSI	Medium Scale Integration
MTU	Magnetic Tape Unit
OCR	Optical Character Recognition
OEM	Original Equipment Manufacturer
OMR	Optical Mark Recognition
PCB	Printed Circuit Board
PIA	Periphery Interface Adapter
PIO	Parallel Input-Output
PISO	Parallel Input Series Output
PL	Programming Language
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RBE	Remote Batch Entry
RJE	Remote Job Entry
ROM	Read Only Memory
RTOS	Real Time Operating System
SIPO	Series Input Parallel Output
SOS	Silicon On Sapphire
TTL	Transistor-Transistor Logic
TTY	Teletypewriter
TWX	Two-Way Transmission
UART	Universal Asynchronous Receiver Transmitter
USRT	Universal Synchronous Receiver Transmitter
VDU	Video Display Unit
VIA	Versatile Interface Adapter

Tabelle 27

			B6 →	0	0	0	0	1	1	1	1
			B5 →	0	0	1	1	0	0	1	1
			B4 →	0	1	0	1	0	1	0	1
B3	B2	B1	B0								
0	0	0	0	NUL	DLE	SP	0	@	P	\	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	"	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	'	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	I	k	{
1	1	0	0	FF	FS	,	<	L	\	l	|
1	1	0	1	CR	GS	-	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^	n	~
1	1	1	1	SI	US	/	?	O	-	o	DEL

ASCII (128)

	B6	B5	B4	B3	B2	B1	B0
Beispiel: codiere 7:	0	1	1	0	1	1	1

Literaturangaben

Mikroprozessor-Bausteine, SAB 8080 System;
Siemens A.G. – München

Digitale techniek; A.J. Dirksen;
De Muiderkring – Bussum

An Introduction to Microcomputers,
Volume 0: The Beginners' Book;
Volume I: Basic Concepts;
Volume II: Some Real Products;
8080 Programming for Logic Design;
Z80 Programming for Logic Design;
6800 Programming for Logic Design;
Some Common Basic Programs (Buch);
A. Osborne, De Muiderkring – Bussum

Sachverzeichnis

A/D-Wandler 47
Adresse 46
Algorithmus 35
Analog-Computer 12

BASIC 203, 204
BCD-Code 17
Befehlszähler 54
Binärspulen 15
Bit 17
Bus 46
Byte 19

Carry-Bit 79
COBOL 203
CPU 48, 174, 188

Datenbus 46
Daten, serielle 140
Daten, parallele 139
Digital-Computer 12
Dual-System 14

EAROM 52
ECOL 203
EPROM 58

Flipflop 61
Floppy Disk 68
Flußdiagramm 35
FORTRAN 203

Hardware 47
Hardware-Interrupt 100

Interface 47

JMP-Befehlscode 96

Kassette 65
Kontaktprellen 138

Load-Befehl 70
Lochkarten 68
Lochstreifen 68

Maschinensprache 119
Maskieren 60
Monitorprogramm 119

Operationscode 51, 119

Programmzähler 54
Puffer 50, 142
Push-Befehl 72

Raster 50
Register 46
Registerinhalt 124
Rotier-Befehl 90

Stack 51
Stackpointer 53
Status-Register 29
Siebensegment-Anzeige 146
Software 47
Software-Interrupt 101
Speicherplatz 36
Sprungbedingung 92
Store-Befehl 70

Taktsignal 56
Tetrade 25

Unterprogramm 37, 44

VDU 47
Vektor 59
Verknüpfungen, logische 33
Verknüpfungsbefehl 110

Wahrheitstabelle 32

Zeitmessung 128
Zero-Page 107, 113

Immerzeel
Mikrocomputer ohne Ballast

M. B. Immerzeel

Die Logik der Mikrocomputer vom binären Rechner bis zum fertigen Programm wird hier ganz einfach dargestellt. Der Autor ist Fachlehrer für Elektronik. Er muß ein guter Lehrer sein, denn der didaktische Aufbau des Buches ist hervorragend. Kein Satz ist zuviel, keiner zuwenig. Wer ein bißchen Verständnis für die Elektronik aufbringt, begreift ganz schnell die Zusammenarbeit von Mensch und Mikrocomputer.
Mit dem Kernsatz „Ein Computer ist in erster Linie eine Rechenmaschine" beginnt der Autor seinen Lehrgang und arbeitet zielbewußt auf den Mikroprozessor 6502 hin, weil dieser in zahlreichen Mikrocomputern eingesetzt ist. Ergänzend, um nicht einseitig zu werden, beschreibt er auch die Mikroprozessoren 6800 und 8080. Wenn das erledigt und begriffen ist, erklärt er die einzelnen Befehle, Adressierungsarten und entwickelt einzelne einfache Standardprogramme, die immer wieder in der praktischen Arbeit angewandt werden können.
Mit einer kurzen Einführung in die Programmiersprache Basic beschließt der Autor sein Buch, das zurecht den Titel trägt: Mikrocomputer ohne Ballast.

ISBN 3-7723-6981-2

Franzis'

Immerzeel **Mikrocomputer ohne Ballast**